Анатолий Хомоненко
Владимир Гофман
Евгений Мещеряков

bhv

Delphi 7

2-е издание

- Визуальная разработка приложений
- Свойства, методы, классы, компоненты
- Локальные и удаленные базы данных
- Работа с электронной почтой и Web-документами

+cd

Наиболее полное руководство

В ПОДЛИННИКЕ®

Анатолий Хомоненко
Владимир Гофман
Евгений Мещеряков

Delphi 7

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2010

УДК 681.3.06
ББК 32.973.26-018.2
Х76

Хомоненко, А. Д.

Х76 Delphi 7 / А. Д. Хомоненко, В. Э. Гофман, Е. В. Мещеряков. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2010. — 1136 с.: ил. + CD-ROM — (В подлиннике)

ISBN 978-5-9775-0425-6

Рассмотрена разработка приложений в Delphi 7 — наиболее популярной версии системы визуального программирования. Описаны основы языка программирования Delphi, а также компоненты, свойства, методы и события, используемые при разработке программ для работы с графикой, мультимедиа, файлами, каталогами и др. Показано применение различных технологий и приемов разработки приложений для работы с базами данных, электронной почтой и Web-документами. Материал сопровождается многочисленными примерами. Во втором издании уделено бóльшее внимание работе с Web-документами, а также обновлен материал по разработке баз данных с помощью технологий dbExpress, ADO и InterBase Express. Компакт-диск содержит листинги программ, приведенных в книге.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.09.09.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 91,59.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0425-6

Оглавление

Предисловие

Книга посвящена популярной версии среды визуальной разработки приложений — Delphi 7. Несмотря на появление более новых версий (например, Delphi NET 2006 и др.), предоставляющих новые, зачастую избыточные возможности, система Delphi 7 по-прежнему находит широкое применение среди разработчиков приложений. На наш взгляд, это объясняется тем, что в Delphi 7 достигнут весьма высокий уровень возможностей, и нет большого количества избыточных и дублирующих друг друга средств.

Авторы не ставили перед собой цель рассмотреть все средства системы, в том числе все новые возможности, что практически невыполнимо в рамках одной книги, поскольку Delphi — это мощная система, предназначенная для быстрой разработки приложений самого разного характера и назначения. Мы стремились показать основные возможности системы Delphi, научить работать с ней, создавать приложения для решения наиболее общих задач, в том числе для работы с базами данных и Интернетом.

В книге описываются основные компоненты, свойства, методы и события. Приводятся примеры, показывающие основные возможности использования большинства средств Delphi 7 с помощью реально работающих программ, которые читатель может использовать в своих разработках.

Система Delphi 7 поставляется в трех версиях: Enterprise, Professional и Personal. Версии различаются своими возможностями, наибольшие возможности предоставляет версия Enterprise, ее мы и использовали при написании книги. Другие версии могут отличаться, например, составом Хранилища объектов или значками объектов, в том числе форм.

Книгу можно рассматривать как исправленное и дополненное произведение В. Гофмана, А. Хомоненко с участием Е. Мещерякова по предыдущей версии системы, которое доработано применительно к работе в среде Delphi 7. Отличия в основном заключаются в освещении ряда новых средств и технологий, появившихся или получивших дальнейшее развитие в системе (технологий dbExpress и ADO работы с базами данных, технологии InterBase Express работы с сервером баз данных InterBase и др.), описании техники работы с новым генератором отчетов Rave 5.0, знакомстве с основами построения и использования Web-служб, уточнении состава визуальных компонентов и др.

Книга ориентирована на широкий круг читателей: от начинающих пользователей до специалистов по программированию.

Далее перечислены сгруппированные по областям применения наиболее важные, на наш взгляд, новые средства и возможности системы Delphi 7.

- Интегрированная среда разработки:
  - появилось окно загрузки с Web-сайтов информации с пояснениями о сообщениях компилятора, стало больше возможностей по управлению предупреждениями компилятора;
  - новые страницы **Indy Intercepts** и **Indy I/O Handlers** Палитры компонентов обеспечивают открытый код компонентов для поддержки межсетевого протокола;
  - новые страницы **IW Standard**, **IW Data**, **IW Client Side** и **IW Control** Палитры компонентов обеспечивают IntraWeb-компоненты для разработки Web-приложений;
  - новая страница **Rave** Палитры компонентов содержит компоненты, предназначенные для генерации отчетов в приложении;
  - завершение кода выполняется быстрее с возможностью просмотра объявлений элементов в коде завершения для любого идентификатора в списке; завершение кода HTML автоматически отображает действительные элементы HTML и атрибуты в редакторе кода; с помощью OpenTools API можно создавать настраиваемые менеджеры завершения кода;
  - можно выполнять установку цвета кода для символов, отображаемых в средствах просмотра кода; отображать в цвете различные типы сообщений о событиях в журнале регистрации; задавать параметры основного цвета и цвета фона вместо цвета сетки в редакторе кода;
  - в отладчике список просмотра для удобства можно компоновать в отдельные группы просмотра, для просмотра введены имена столбцов и строк, добавлен переключатель для управления индивидуальными просмотрами;
  - можно задавать параметры редактора для кода различных типов, таких как Pascal, C++, C#, HTML и XML; отображать символы пробела и табуляции в редакторе кода; редактировать шаблоны кода.
- Web-технологии:
  - в Delphi входит IntraWeb из состава AtoZed Software; его можно использовать для разработки серверных Web-приложений с помощью стандартных средств формы, а также для разработки страниц приложений Web Broker и WebSnap;
  - Delphi поддерживает Apache 2 как целевой тип для Web Broker, WebSnap и SOAP (Simple Object Access Protocol); вместо интерфейса Win-CGI рекомендуется использование CGI, ISAPI/NSAPI или Apache в качестве целевого типа;
  - новый UDDI-обозреватель позволяет размещать и импортировать WSDL-документы, учитываемые в журналах регистрации UDDI;

- новые классы и интерфейсы позволяют читать или вставлять заголовки в конверты SOAP, передающие сообщения между клиентами и серверами; клиентские и серверные приложения Web-служб получили возможность управлять прикрепленными сообщениями;
- предоставляются большие возможности по управлению взаимодействием между вызываемым интерфейсом и документом WSDL на основе использования публикуемых событий;
- новый интерфейс IRIOAccess позволяет осуществлять доступ к удаляемому объекту интерфейса, который реализует вызываемый интерфейс.

♦ Технологии работы с базами данных:

- драйверы dbExpress обновлены для поддержки работы с Informix SE, Oracle 9i, DB2 7.2, InterBase 6.5 и MySQL 3.23.49; имеется новый драйвер для MS SQL 2000;
- имеется несколько новых и измененных компонентов для работы с базами данных; вместо SQL Links для доступа к базам данных SQL Server рекомендуется использовать dbExpress;
- можно задать специальный модуль данных SOAP в сервере приложений, имеющий многочисленные модули данных (свойство `SOAPServerIID` или добавление интерфейса модулей данных в конец URL); использовать компонент-соединение SOAP для вызова расширений интерфейса серверов приложения (свойство `SOAPServerIID` и метод `GetSOAPServer`).

♦ Actions (создание интерфейса пользователя):

- добавлены новые компоненты (на странице **Additional** Палитры компонентов), предназначенные для синхронизации элементов управления: `ActionManager` (менеджер действий), `ActionMainMenuBar` (главное меню действий), `ActionToolBar` (панель действий);
- ряд новых стандартных действий добавлен к VCL, включая действия: с файлами, по поиску, по работе с расширенным текстовым редактором, табуляции, диалоговые, управления списком и по работе с URL.

♦ Технология CORBA: появилась новая версия компилятора IDL2PAS для разработки CORBA-приложений, рекомендуемая для использования вместо старых средств, интегрированных с поддержкой технологии COM.

♦ Библиотека визуальных компонентов VCL:

- добавлен ряд новых компонентов (`TLabeledEdit`, `TValueListEditor`, `TComboBoxEx` и `TColorBox`), представляющих развитые элементы управления для работы с редактором с меткой, пользовательской сеткой, со списком образов и с выбранными цветами;
- ряд элементов управления имеет улучшенные свойства; системные цвета отсортированы для облегчения их поиска; к стандартным шестнадцати цветам добавлены еще четыре цвета.

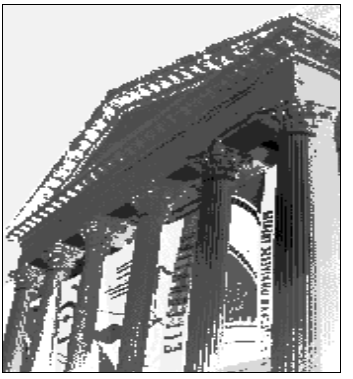
- Библиотека времени выполнения RTL:
  - ряд функций перемещен из других модулей в модуль `System`, в то же время много функций из этого модуля перемещены в новый модуль `Variants`;
  - модуль `System` содержит новые подпрограммы для работы с динамическими массивами (`DynArrayClear` и `DynArraySetLength`), добавлен `IInterface` для работы с отличными от COM интерфейсами.
- Пользовательские варианты:
  - разрешается определять пользовательские типы данных для вариантов (Variants).
- Межплатформенная разработка приложений:
  - можно разрабатывать приложения, функционирующие под управлением Windows и Linux, с помощью библиотеки CLX (межплатформенный вариант библиотеки VCL);
  - для создания приложений, использующих CLX, в Хранилище объектов имеются объекты `CLX Application` и `CLX About Box`.

Во втором издании книги Delphi 7 переработаны материалы по механизму действия *(см. главу 5)*, появились материалы по работе с Web-документами *(см. главу 34)*, заново написаны *главы 21*, *22* и *27*[1]. Для лучшего представления материала переработан подраздел *главы 16*, посвященный характеристике приложения для работы с базами данных.

Авторы[2]

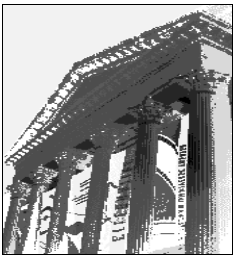
[1] Авторы выражают признательность Владимиру Никифорову за подготовку для первого издания книги материалов *глав 21*, *22* и *27*, посвященных технологиям dbExpress, ADO и InterBase Express.

[2] Материалы *глав 21*, *22*, *25* и *27* и *разд. "Работа с Web-документами" главы 34* подготовлены А. Хомоненко; *главы 30—33* — Е. Мещеряковым и А. Хомоненко при участии В. Гофмана. Остальные главы подготовлены В. Гофманом и А. Хомоненко совместно. Исправление и дополнение материалов книги применительно к версии Delphi 7, переработку *глав 5* и *16*, а также подготовку компакт-диска выполнил А. Хомоненко.



ЧАСТЬ I

Введение в Delphi 7



ГЛАВА 1

Среда Delphi 7

Прикладные программы, или приложения, Delphi создаются в *интегрированной среде разработки* (IDE — Integrated Development Environment). Пользовательский интерфейс этой среды служит для организации взаимодействия с программистом и включает в себя ряд окон, содержащих различные элементы управления. С помощью средств интегрированной среды разработчику удобно проектировать интерфейсную часть приложения, а также писать программный код и связывать его с элементами управления. В интегрированной среде разработки проходят все этапы создания приложения, включая отладку.

Интегрированная среда разработки Delphi 7 представляет собой многооконную систему. Вид интегрированной среды разработки (пользовательский интерфейс) может различаться в зависимости от настроек. После загрузки интерфейс Delphi 7 выглядит так, как показано на рис. 1.1, и первоначально включает шесть окон:

- главное окно (**Delphi 7 — Project1**);
- окно Обозревателя дерева объектов (**Object TreeView**);
- окно Инспектора объектов (**Object Inspector**);
- окно Формы, или Конструктора формы (**Form1**);
- окно Редактора кода (**Unit1.pas**).
- окно Проводника кода (**Exploring Unit1.pas**).

Последние два окна находятся позади окна Формы, причем окно Проводника кода пристыковано слева к окну Редактора кода, поэтому оба этих окна имеют общий заголовок **Unit1.pas**.

На экране кроме указанных окон могут присутствовать и другие окна, отображаемые при вызове соответствующих средств, например, окно Редактора изображений (Image Editor). Окна Delphi можно перемещать, изменять их размеры и убирать с экрана (кроме главного окна), а также состыковывать между собой.

Несмотря на наличие многих окон, Delphi является однодокументной средой и позволяет одновременно работать только с одним приложением (проектом приложения). На-

звание проекта приложения выводится в строке заголовка главного окна в верхней части экрана.

При сворачивании главного окна сворачивается весь интерфейс Delphi и, соответственно, все открытые окна; при закрытии главного окна работа с Delphi прекращается. Главное окно Delphi включает:

- главное меню;
- панели инструментов;
- Палитру компонентов.

Главное меню содержит обширный набор команд для доступа к функциям Delphi, основные из которых рассматриваются при изучении связанных с этими командами операций.

Панели инструментов находятся под главным меню в левой части главного окна и содержат пятнадцать кнопок для вызова наиболее часто используемых команд главного меню, например, **File | Open** (Файл | Открыть) или **Run | Run** (Выполнение | Выполнить).

Вызвать многие команды главного меню можно также с помощью комбинаций клавиш, указываемых справа от названия соответствующей команды. Например, команду **Run | Run** можно вызвать с помощью клавиши <F9>, а команду **View | Units** (Просмотр | Модули) — с помощью комбинации клавиш <Ctrl>+<F12>.

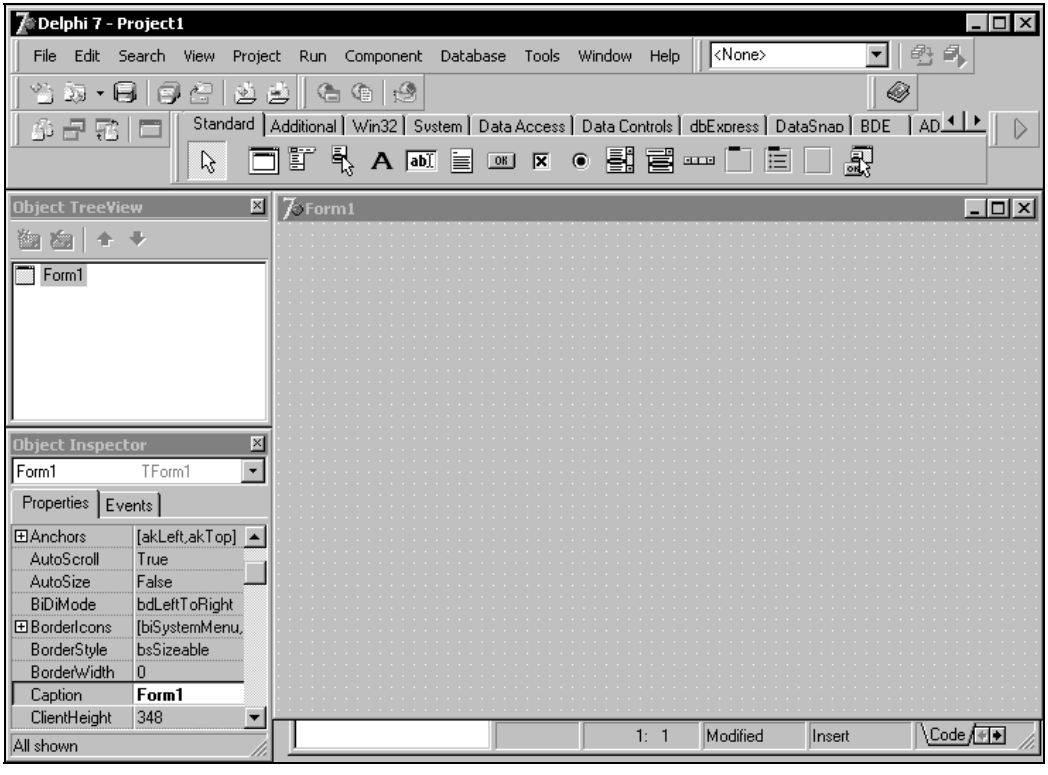


Рис. 1.1. Вид интегрированной среды разработки

Всего имеется 6 панелей инструментов:

- **Standard** (Стандартная);
- **View** (Просмотр);
- **Debug** (Отладка);
- **Custom** (Пользователь);
- **Desktop** (Рабочий стол);
- **Internet** (Интернет).

Отображением панелей инструментов и настройкой кнопок на них можно управлять. Эти действия выполняются с помощью контекстного меню панелей инструментов, вызываемого щелчком правой кнопки мыши при размещении указателя в области панелей инструментов или главного меню.

С помощью контекстного меню можно также управлять видимостью Палитры компонентов (**Component Palette**).

Более широкие возможности по настройке панелей инструментов и главного меню предоставляет показанное на рис. 1.2 диалоговое окно **Customize** (Индивидуальная настройка), вызываемое одноименной командой контекстного меню панелей инструментов. С его помощью можно скрыть или отобразить требуемую панель инструментов, изменить состав кнопок на ней, а также выбрать режим отображения всплывающих подсказок для кнопок.

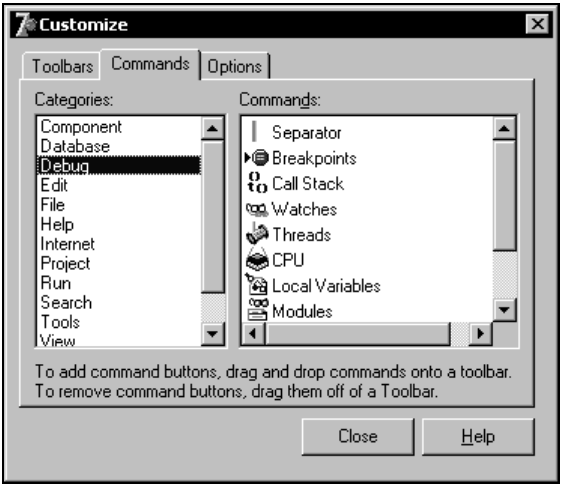


Рис. 1.2. Диалоговое окно индивидуальной настройки панелей инструментов

Палитра компонентов находится под главным меню в правой части главного окна и содержит множество компонентов, размещаемых в создаваемых формах. *Компоненты* являются своего рода строительными блоками, из которых конструируются формы приложения. Все компоненты разбиты на группы, каждая из которых в Палитре компонентов располагается на отдельной странице, а сами компоненты представлены значками. Нужная страница Палитры компонентов выбирается щелчком мыши на ее значке.

Первоначально Палитра компонентов имеет следующий набор страниц:

- **Standard** — стандартная;
- **Additional** — дополнительная;

- **Win32** — 32-разрядного интерфейса Windows;
- **System** — доступа к системным функциям;
- **Data Access** — работы с информацией из баз данных;
- **Data Controls** — создания элементов управления данными;
- **dbExpress** — доступа к SQL-серверам;
- **DataSnap** — создания многоуровневых приложений баз данных;
- **BDE** — доступа к данным с помощью BDE;
- **ADO** — связи с базами данных с использованием объектов данных ActiveX;
- **Interbase** — обеспечения непосредственного доступа к одноименной базе данных;
- **WebServices** — создания клиентских приложений, использующих Web-сервис с помощью технологии SOAP;
- **InternetExpress** — создания приложений InternetExpress, которые являются одновременно Web-сервером и клиентом распределенной базы данных;
- **Internet** — создания приложений Web-сервера для Интернета;
- **WebSnap** — создания приложений Web-серверов;
- **Decision Cube** — многомерного анализа;
- **Dialogs** — создания стандартных диалоговых окон;
- **Win 3.1** — интерфейса Windows 3.*x*;
- **ActiveX** — компонентов ActiveX;
- **RAVE** — генерации отчетов в приложении;
- **Indy Clients** — платформо-независимые компоненты Интернета для клиента;
- **Indy Servers** — платформо-независимые компоненты Интернета для сервера;
- **Indy Intercepts** — платформо-независимые компоненты Интернета для обработки прерываний при кодировании/раскодировании и преобразовании передаваемой информации;
- **Indy I/O Handlers** — платформо-независимые компоненты Интернета для управления вводом/выводом;
- **Indy Misc** — дополнительные платформо-независимые компоненты Интернета (обработки, кодирования и декодирования данных);
- **COM+** — управления одноименными объектами;
- **InterBase Admin** — управления доступом к одноименной базе данных;
- **IW Standard** — стандартная для работы в Интернете;
- **IW Data** — создания элементов управления при работе с базами данных в Интернете;
- **IW Client Side** — для обеспечения работы в Интернете со стороны клиента;
- **IW Control** — управления работой в Интернете;
- **Servers** — оболочки VCL для общих серверов COM (Microsoft Office 2000 или 97).

Более подробно основные из указанных компонентов будут рассматриваться по ходу изложения.

Палитру компонентов можно настраивать с помощью диалогового окна **Palette Properties** (Свойства палитры). Это окно (рис. 1.3) вызывается командой **Properties** (Свойства) контекстного меню Палитры компонентов или командой **Component | Configure Palette** (Компонент | Настройка палитры) главного меню. Окно позволяет выполнять такие операции, как удаление, добавление отдельных компонентов и перемещение их на другое место, а также добавление, удаление или перемещение страниц компонентов.

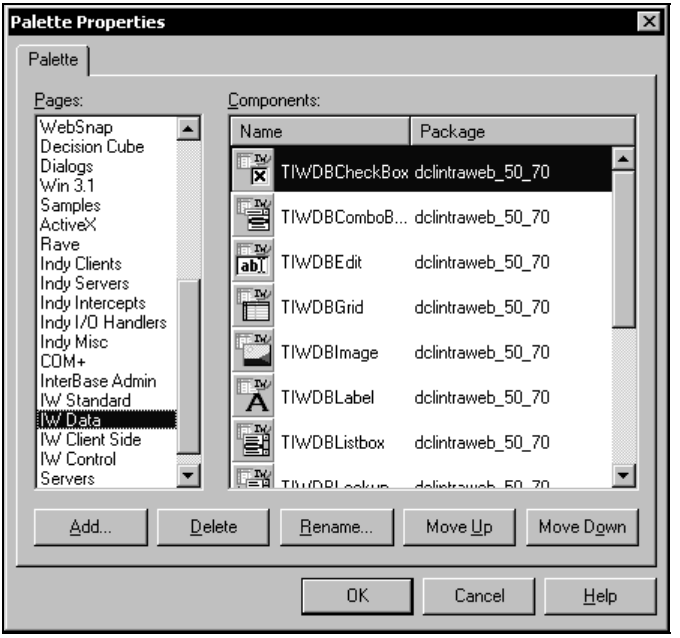


Рис. 1.3. Диалоговое окно свойств Палитры компонентов

В списке **Pages** диалогового окна **Palette Properties** содержатся названия страниц, в списке **Components** — названия компонентов выбранной страницы. С помощью кнопок, расположенных под списками, задаются следующие действия:

- добавить (**Add**), удалить (**Delete**) или переименовать (**Rename**) страницу;
- переставить страницу или компонент на позицию выше (**Move Up**) или ниже (**Move Down**);
- скрыть компонент (**Hide**).

Замечание

Удаление страницы возможно только в случае, если она не содержит компонентов, т. е. является пустой.

При выборе в списке **Components** компонента название кнопки **Delete** (Удалить) изменяется на **Hide** (Скрыть).

При скрытии компонента он удаляется из страницы, однако модуль с описанием его класса остается на диске, и компонент может быть установлен.

Установка компонента в окне **Palette Properties** не выполняется. Вопросы, связанные с созданием и установкой новых компонентов, будут отдельно рассмотрены в соответствующей главе.

Окно Формы (или Конструктора формы) первоначально находится в центре экрана и имеет заголовок **Form1**. В нем выполняется проектирование формы, в процессе которого в форму из Палитры компонентов помещаются необходимые компоненты. При этом проектирование заключается в визуальном конструировании формы, а действия разработчика похожи на работу в среде простого графического редактора. Сам Конструктор формы во время ее проектирования остается как бы "за кадром", и разработчик имеет дело непосредственно с формой, поэтому часто окно Конструктора формы также называют окном Формы или просто "формой".

Окно Редактора кода (**Unit1.pas**) после запуска системы программирования находится под окном Формы и почти полностью перекрывается им. Редактор кода представляет собой обычный текстовый редактор, с помощью которого можно редактировать текст модуля и другие текстовые файлы приложения, например, файл проекта. Каждый редактируемый файл находится в окне Редактора кода на отдельной странице, доступ к которой осуществляется щелчком на соответствующем значке. Первоначально в окне Редактора кода на странице **Code** содержится одна вкладка **Unit1** исходного кода модуля формы `Form1` разрабатываемого приложения.

В Delphi 7 Редактор кода поддерживает также просмотр и редактирование других элементов приложения. Для этого используются страницы:

- **Diagram** — отображение и настройка взаимосвязей между визуальными и невизуальными компонентами;
- **HTML Script** — просмотр документа HTML и текста JavaScript, сгенерированных с помощью компонента типа `TAdapterPageProducer`;
- **HTML Result** — просмотр документа HTML, сгенерированного на основе HTML-шаблона;
- **Preview** — просмотр документа HTML в окне обозревателя;
- **XML Tree** — отображение документа XML или модуля Web-страницы в окне обозревателя;
- **XSL Tree** — отображение документа XSL или модуля Web-страницы.

В окне Редактора кода всегда присутствует страница **Code**, а остальные страницы требуют соответствующей настройки.

Переключаться между окнами Формы и Редактора кода удобно с помощью клавиши <F12>.

Окно Проводника кода (**Exploring Unit1.pas**) пристыковано слева к окну Редактора кода. В нем в виде дерева отображаются все объекты модуля формы, например переменные и процедуры (рис. 1.4). В окне Проводника кода удобно просматривать объек-

ты приложения и быстро переходить к нужным объектам, что особенно важно для больших модулей. Окно Проводника кода открывается командой **Code Explorer** (Проводник кода) меню **View** (Просмотр).

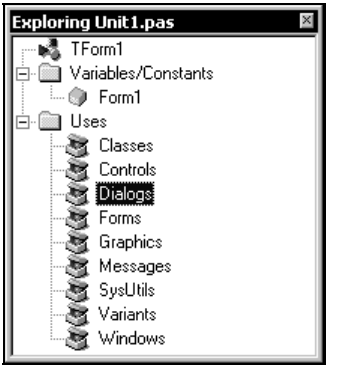


Рис. 1.4. Окно Проводника кода

Для настройки Проводника кода служит показанное на рис. 1.5 окно **Explorer Options** (Параметры проводника), открываемое командой **Properties** (Свойства) контекстного меню Проводника кода. С помощью этого окна можно, например, управлять отображением объектов категорий, содержащихся в списке **Explorer categories** (Категории просмотра). Чтобы окно Проводника кода по умолчанию отсутствовало, нужно сбросить флажок **Automatically show Explorer** (Автоматически отображать Проводник).

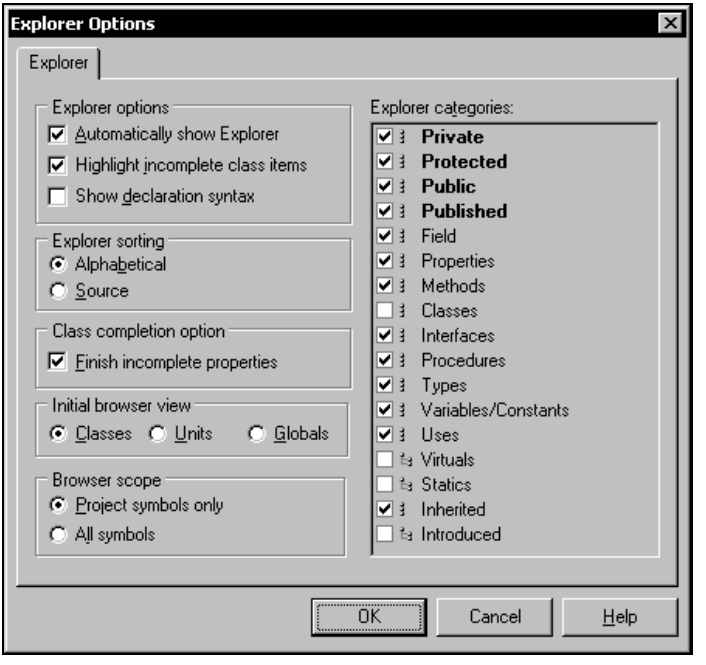


Рис. 1.5. Окно параметров Проводника кода

Окно Обозревателя дерева объектов после запуска системы находится под главным окном и отображает древовидную структуру объектов текущей формы (первоначально `Form1`). Это окно удобно использовать в случае форм, служащих для обработки баз данных, т. к. оно позволяет изменять связи между компонентами, например, переназначить таблице источник данных другой таблицы. На рис. 1.6 показана структура объектов формы `Form1`, предназначенной для работы с таблицей базы данных.

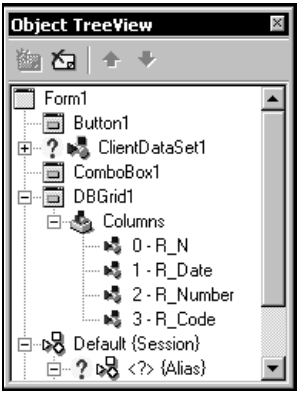


Рис. 1.6. Окно Обозревателя дерева объектов

Окно Инспектора объектов находится под окном Обозревателя дерева объектов в левой части экрана и отображает свойства и события объектов для текущей формы `Form1`. Его можно открыть командой **View | Object Inspector** (Просмотр | Инспектор объектов) или нажатием клавиши <F11>.

Окно Инспектора объектов имеет две страницы: **Properties** (Свойства) и **Events** (События).

Страница **Properties** отображает информацию о текущем (выбранном) компоненте в окне Формы и при проектировании формы позволяет удобно изменять многие свойства компонентов.

Страница **Events** определяет процедуру, которую компонент должен выполнить при возникновении указанного события. Если для какого-либо события задана такая процедура, то в процессе выполнения приложения при возникновении этого события процедура вызывается автоматически. Такие процедуры служат для обработки соответствующих событий, поэтому их называют *процедурами — обработчиками событий* или просто *обработчиками*. Отметим, что события также являются свойствами, которые указывают на свои обработчики.

В конкретный момент времени Инспектор объектов отображает свойства и события текущего (выбранного) компонента, имя и тип которого отображаются в списке под заголовком окна Инспектора объектов. Компонент, расположенный в форме, можно выбрать щелчком мыши на нем или в списке Инспектора объектов. У каждого компонента есть набор свойств и событий, определяющих его особенности.

Инспектор объектов позволяет группировать свойства и события по категориям или в алфавитном порядке. Свойства (и их значения) отображаются различными цветами.

В Инспекторе объектов содержатся и свойства, предназначенные только для чтения. Кроме того, можно настраивать свойства Инспектора объектов.

По умолчанию Инспектор объектов отображает названия свойств и событий в алфавитном порядке (см. рис. 1.1). Отображение их по категориям выполняется командой **Arrange | by Category** (Расположить | По категориям) контекстного меню Инспектора объектов, при этом Инспектор объектов принимает вид, показанный на рис. 1.7. Командой **Arrange | by Name** (Расположить | По имени) восстанавливается расположение по алфавиту.

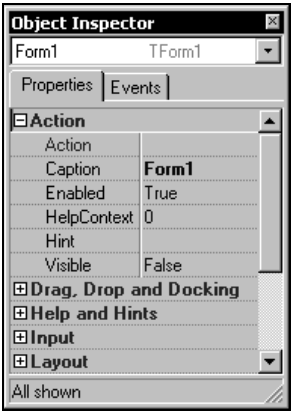


Рис. 1.7. Расположение свойств и событий по категориям

По умолчанию Инспектор объектов отображает все свойства и события объектов. Можно отключить/включить отображение некоторой категории, убрав/установив отметку в соответствующем пункте (например, **Action**) подменю команды **View** контекстного меню.

Для настройки вида Инспектора объектов служит окно **Object Inspector Properties** (Свойства Инспектора объектов) (рис. 1.8), открываемое командой **Properties** контекстного меню Инспектора объектов. С его помощью можно выбрать, например, цвет для отображения имен (**Name**) и значений (**Value**) свойств.

Delphi поддерживает технологию *Dock-окон*, которые могут стыковаться (соединяться) друг с другом с помощью мыши. Такими окнами являются инструментальные (недиалоговые) окна интегрированной среды разработки, в том числе окна Инспектора объектов и Проводника кода. Состыкованные окна удобно, например, перемещать по экрану или изменять их размеры.

Для соединения двух окон следует с помощью мыши поместить одно из них на другое и после изменения вида рамки перемещаемого окна отпустить его, после чего это окно автоматически пристыкуется сбоку ко второму окну. Разделение окон выполняется перемещением пристыкованного окна за двойную линию, размещенную под общим заголовком. После соединения окна представляют собой одно общее окно, разделенное на несколько частей. При стыковке/отстыковке окно изменяет свое название. Так, окно Проводника кода, состыкованное с окном Редактора кода, имеет общее с ним название, например, **Unit1.pas**, в то время как при отстыковке название изменяется на **Exploring**

Unit1.pas. Окна Инспектора объектов и Обозревателя дерева объектов при стыковке объединяют свои названия (названия всех окон указываются через запятую).

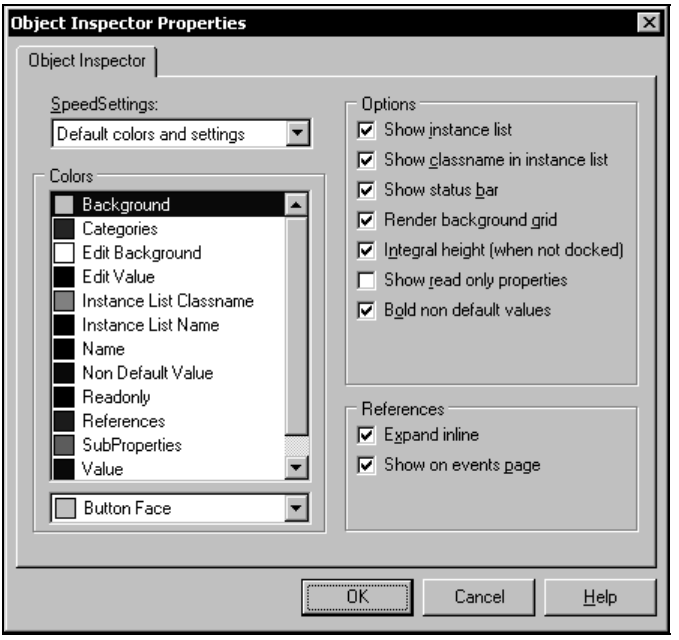


Рис. 1.8. Окно свойств Инспектора объектов

Можно запретить стыковку окна, убрав отметку **Dockable** (Стыкуемое) в контекстном меню окна. По умолчанию эта отметка включена, и окно является стыкуемым.

Для окон Инспектора объектов и Обозревателя дерева объектов можно установить режим **Stay on Top** (Расположить наверху), расположив их поверх других окон. Это выполняется включением одноименной отметки в контекстном меню.

Скрытое окно вызывается на экран командой меню **View**. Например, окно Проводника кода выводится на экран командой **View | Code Explorer**.

Характеристика проекта

В этом разделе рассматриваются: состав проекта, файл проекта, файлы формы, файлы модулей, файл ресурсов и параметры проекта.

Состав проекта

Создаваемое в среде Delphi приложение состоит из нескольких элементов, объединенных в *проект*. В состав проекта входят следующие элементы (в скобках указаны расширения имен файлов):

- код проекта (dpr);
- описания форм (dfm — для Windows, xfm — кроссплатформенный вариант);

- модули и модули форм (pas);
- параметры проекта (dof — для Windows, kof — для Linux);
- параметры среды (cfg);
- описание ресурсов (res).

Взаимосвязи между отдельными частями (файлами) проекта показаны на рис. 1.9.

Кроме приведенных файлов, автоматически могут создаваться и другие файлы, например, резервные копии файлов: ~dp — для файлов с расширением dpr; ~pa — для файлов с расширением pas.

При запуске Delphi автоматически создается новый проект с именем `Project1`, отображаемым в заголовке главного окна Delphi. Этот проект имеет в своем составе одну форму `Form1`, название которой видно в окне Формы. Разработчик может изменить предлагаемое по умолчанию имя проекта, а также установить параметры среды таким образом, что после загрузки Delphi будет автоматически загружаться приложение, разработка которого выполнялась в последний раз.

Обычно файлы проекта располагаются в одном каталоге. Поскольку даже относительно простой проект включает в себя достаточно много файлов, а при добавлении к проекту новых форм количество этих файлов увеличивается, для каждого нового проекта целесообразно создавать отдельный каталог, где и сохранять все файлы проекта.

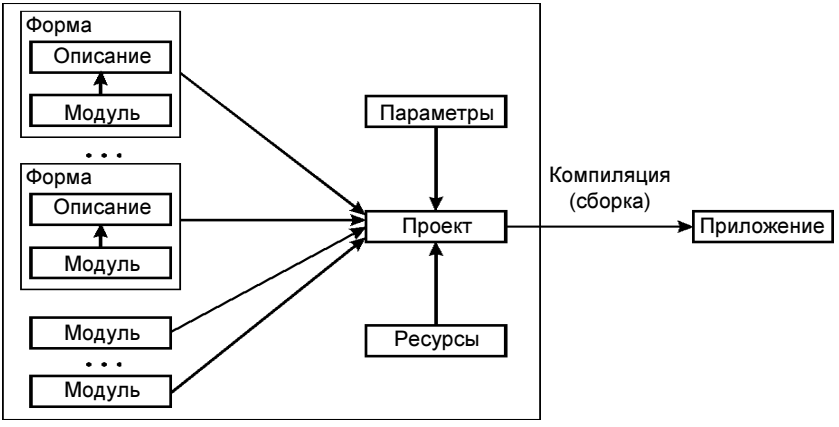


Рис. 1.9. Связь между файлами проекта

Файл проекта

Файл проекта является центральным файлом проекта и представляет собой собственно программу. Для приложения, имеющего в составе одну форму, файл проекта имеет следующий вид:

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
```

```
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Имя проекта (программы) совпадает с именем файла проекта и указывается при сохранении этого файла на диске, первоначально это имя `Project1`. То же имя имеют файлы ресурсов и параметров проекта, при переименовании файла проекта данные файлы переименовываются автоматически.

Сборка всего проекта выполняется при компиляции файла проекта. При этом имя создаваемого приложения (exe-файл) или динамически загружаемой библиотеки (dll-файл) совпадает с названием файла проекта.

В дальнейшем мы будем подразумевать, что создается приложение, а не динамически загружаемая библиотека.

В разделе `uses` указывается имя подключаемого модуля `Forms`, который является обязательным для всех приложений, имеющих в своем составе формы. Кроме того, в разделе `uses` перечисляются подключаемые модули всех форм проекта, первоначально это модуль `Unit1` формы `Form1`.

Директива `$R` подключает к проекту файл ресурсов, имя которого по умолчанию совпадает с именем файла проекта. Поэтому вместо имени файла ресурса указан символ `*`. Кроме этого файла разработчик может подключить к проекту и другие ресурсы, самостоятельно добавив директивы `$R` и указав в них соответствующие имена файлов ресурсов.

Замечание

Ресурсы, указанные в директиве `$R`, подключаются к проекту при его компиляции и сборке, увеличивая размер файла приложения. Поэтому желательно подключать таким способом только относительно небольшие в смысле затрат памяти ресурсы, например, значки или курсоры. Большие растровые изображения лучше подключать динамически, используя соответствующие методы, например, `LoadFromFile`.

Программа проекта содержит всего три инструкции, выполняющие инициализацию приложения, создание формы `Form1` и запуск приложения. Эти инструкции будут рассмотрены в следующих главах.

При выполнении разработчиком каких-либо операций с проектом Delphi формирует код файла проекта автоматически. Например, при добавлении новой формы в файл проекта добавляются две строки кода, относящиеся к этой форме, а при исключении формы из проекта эти строки автоматически исключаются. При необходимости программист может вносить изменения в файл проекта самостоятельно, однако подобные действия могут разрушить целостность проекта и поэтому обычно выполняются только опытными программистами. Отметим, что некоторые операции, например, создание обработчика события для объекта `Application`, системой Delphi автоматически не выполняются и требуют самостоятельного кодирования в файле проекта.

Отображение кода файла проекта в окне Редактора кода задается командой **Project | View Source** (Проект | Просмотр источника).

В файле проекта для многих приложений имеется похожий код, поэтому в дальнейшем при рассмотрении большинства приложений содержимое этого файла нами не приводится.

Файлы формы

Для каждой формы в составе проекта автоматически создаются файл описания формы (расширение dfm) и файл модуля формы (расширение pas).

Файл описания формы является ресурсом Delphi и содержит характеристики формы и ее компонентов. Разработчик обычно управляет этим файлом через окна Формы и Инспектора объектов. При конструировании формы в файл описания автоматически вносятся соответствующие изменения.

Замечание

Файл описания формы является ресурсом Delphi, поскольку он разработан именно для этой среды и интерпретируется ею при создании формы приложения.

Содержимое файла описания формы определяет ее вид. При необходимости можно отобразить этот файл на экране в текстовом виде, что выполняется командой **View as Text** (Просмотреть как текст) контекстного меню формы. При этом окно Формы пропадает с экрана, а содержимое файла описания формы открывается в окне Редактора кода и доступно для просмотра и редактирования. В качестве примера далее приведен текст файла описания простой формы: она содержит одну кнопку `Button1`, для которой создан обработчик события `OnClick`.

```
object Form1: TForm1
  Left = 192
  Top = 107
  Width = 544
  Height = 375
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 88
    Top = 120
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
    OnClick = Button1Click
  end
end
```

Отметим, что в начальной (пустой) форме типа `TForm1` отсутствуют строки, относящиеся к кнопке `Button1` (выделены полужирным начертанием).

Из приведенного примера видно, что файл описания содержит перечень всех объектов формы, включая саму форму, а также свойства этих объектов. Для каждого объекта указывается его тип; для формы ее тип (класс) `TForm1` описывается в модуле этой формы. Если в строчке `Caption = 'Form1'`, определяющей заголовок формы, вместо `Form1` ввести, например, текст `Первая форма`, то заголовок формы изменится на новый. Однако на практике подобные действия обычно выполняются в окне Инспектора объектов.

Повторное открытие окна формы выполняется командой **View | Forms** (Просмотр | Формы) или комбинацией клавиш <Shift>+<F12>, после чего открывается диалоговое окно **View Form** (Просмотр форм), в списке которого и выбирается нужная форма (рис. 1.10).

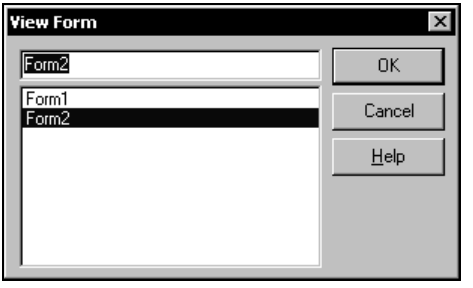


Рис. 1.10. Выбор файла описания формы

Одновременно можно отобразить на экране несколько форм. Для закрытия того или иного окна Формы нужно выполнить команду **File | Close** (Файл | Закрыть) или щелкнуть мышью на кнопке закрытия соответствующего окна.

Файл модуля формы содержит описание класса формы. Для пустой формы, добавляемой к проекту по умолчанию, файл модуля формы содержит следующий код:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
private
 { Private declarations }
public
 { Public declarations }
end;
var
  Form1: TForm1;
implementation
  {$R *.dfm}
end.
```

Delphi автоматически создает файл модуля формы при добавлении новой формы. По умолчанию к проекту добавляется новая форма типа `TForm`, не содержащая компонентов.

В разделе `interface` модуля формы содержится описание класса формы, а в разделе `implementation` — подключение к модулю директивой `$R` визуального описания соответствующей формы. При размещении в форме компонентов, а также при создании обработчиков событий в модуль формы вносятся соответствующие изменения. При этом часть этих изменений Delphi выполняет автоматически, а часть пишет разработчик. Обычно все действия разработчика, связанные с программированием, выполняются именно в модулях форм.

Тексты файлов модулей форм отображаются и редактируются с помощью Редактора кода. Открыть файл модуля формы можно в стандартном окне открытия файла (команда **File | Open** (Файл | Открыть)) или в диалоговом окне **View Unit** (рис. 1.11), открываемом командой **View | Units** (Просмотр | Модули) или нажатием комбинации клавиш <Ctrl>+<F12>. В окне открытия файла модуля формы можно выбрать также файл проекта. После выбора нужного модуля (или проекта) и нажатия кнопки **OK** его текст появляется на отдельной странице Редактора кода.

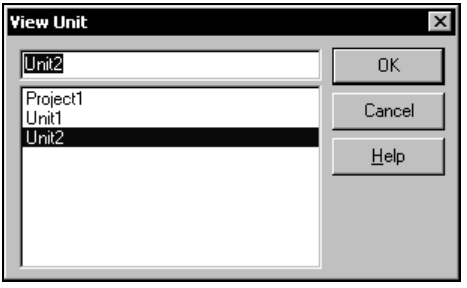


Рис. 1.11. Открытие файла модуля формы

Отметим, что оба файла каждой формы (описания и модуля) имеют одинаковые имена, отличные от имени файла проекта, хотя у имени файла проекта и другое расширение.

При компиляции модуля автоматически создается файл с расширением dcu (dpu — для Linux), который содержит откомпилированный код модуля. Этот файл можно удалить из каталога, в котором находятся все файлы проекта, — Delphi снова создаст этот файл при следующей компиляции модуля или проекта. Смысл создания названных файлов в Delphi состоит в том, что практически можно собрать проект, используя только dcu-файлы без pas-файлов.

Файлы модулей

Кроме модулей в составе форм, при программировании можно использовать и *отдельные модули*, не связанные с какой-либо формой. Они оформляются по обычным правилам языка Object Pascal и сохраняются в отдельных файлах. Для подключения модуля его имя указывается в разделе `uses` того модуля или проекта, который использует средства этого модуля.

В отдельном модуле можно (даже полезно) размещать процедуры, функции, константы и переменные, общие для нескольких модулей проекта.

Файл ресурсов

При первом сохранении проекта автоматически создается файл ресурсов (расширение res) с именем, совпадающим с именем файла проекта. Файл ресурсов может содержать следующие ресурсы:

- значки;
- растровые изображения;
- курсоры.

Перечисленные компоненты являются ресурсами Windows, поскольку они разработаны и интерпретируются в соответствии со стандартами этой операционной системы.

Первоначально файл ресурсов содержит значок проекта, которым по умолчанию является изображение факела. В дальнейшем его можно изменить или заменить.

Для работы с файлами ресурсов в состав Delphi включен графический редактор Image Editor версии 3.0, вызываемый командой **Tools | Image Editor** (Средства | Редактор изображений). На рис. 1.12 показан вид окна Редактора изображений с загруженным файлом ресурсов Project1.res для редактирования значка приложения.

Файл ресурсов имеет иерархическую структуру, в которой ресурсы разбиты на группы, и каждый ресурс имеет уникальное в пределах группы имя. Имя ресурса задается при его создании и в последующем используется в приложении для доступа к этому ресурсу. Значок проекта находится в группе `Icon` и по умолчанию имеет имя `MAINICON`.

Рис. 1.12. Окно Редактора изображений

Кроме файла с расширением res, объединяющего несколько ресурсов, редактор Image Editor также позволяет работать с файлами, содержащими следующие ресурсы (в скобках указано расширение имени файла):

- значки компонентов (dcr);
- растровые изображения (bmp);
- значки приложений (ico);
- курсоры (cur).

Параметры проекта

Для установки параметров проекта используется окно параметров проекта (**Project Options**), открываемое командой **Project | Options** (Проект | Параметры) или нажатием комбинации клавиш <Shift>+<Ctrl>+<F11>. Параметры разбиты на группы, каждая из которых располагается в окне параметров проекта на своей странице (рис. 1.13).

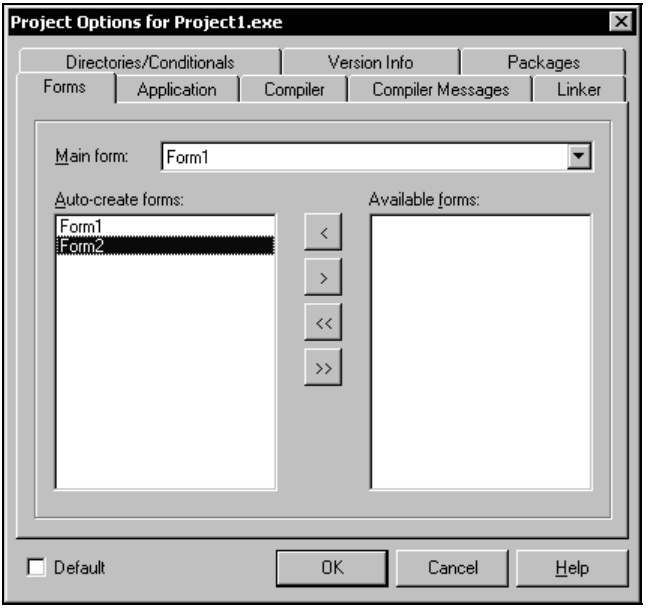


Рис. 1.13. Окно параметров проекта

После установки отдельных параметров Delphi автоматически вносит нужные изменения в соответствующие файлы проекта. Так, параметры из страниц **Forms** и **Application** вносятся в файлы проекта и ресурсов, а параметры из страниц **Compiler** и **Linker** — в файл параметров проекта.

Далее для примера приводятся фрагменты файла параметров проекта.

```
[Compiler]
A=8
B=0
C=1
...
[Version Info Keys]
CompanyName=
```

```
FileDescription=
FileVersion=1.0.0.0
...
```

Как видите, файл параметров проекта представляет собой текстовый файл, в котором построчно записаны параметры и их значения.

Компиляция и выполнение проекта

В процессе компиляции проекта создается готовый к использованию файл, которым может быть *приложение* (расширение exe) или *динамически загружаемая библиотека* (расширение dll). Как говорилось ранее, мы будем рассматривать только приложения. Имя приложения, получаемого в результате компиляции, совпадает с именем файла проекта, а само приложение является автономным и не требует для своей работы дополнительных файлов Delphi.

Замечание

Если в процессе выполнения приложения динамически используются другие файлы, например, изображения или файлы справки, то эти файлы должны быть в наличии.

При создании приложений, работающих с базами данных, необходимы файлы, составляющие базу данных, а также процессор баз данных.

Компиляция выполняется вызовом команды **Project | Compile *<Project1>*** (Проект | Компилировать *<Проект1>*) или нажатием комбинации клавиш <Ctrl>+<F9>. В команде содержится имя проекта, разработка которого осуществляется в настоящий момент (первоначально `Project1`). При сохранении проекта под другим именем соответственно должно быть изменено и имя проекта в команде меню.

Скомпилировать проект для получения приложения можно на любой стадии разработки проекта. Это удобно для проверки вида и правильности функционирования отдельных компонентов формы, а также для тестирования фрагментов создаваемого кода. При компиляции проекта выполняются действия, приведенные далее.

- Компилируются файлы всех модулей, содержимое которых изменилось со времени последней компиляции. В результате для каждого файла с исходным текстом модуля создается файл с расширением dcu. Если исходный текст модуля по каким-либо причинам недоступен компилятору, то он не перекомпилируется.
- Если в модуль были внесены изменения, то перекомпилируется не только этот модуль, но и модули, использующие его с помощью директивы `uses`.
- Перекомпиляция модуля происходит также при изменениях объектного файла (расширение obj) или подключаемого файла (расширение inc), используемых данным модулем.
- После компиляции всех модулей проекта компилируется файл проекта и создается исполняемый файл приложения с именем файла проекта.

Помимо компиляции, может быть выполнена также *сборка* проекта. При сборке компилируются все файлы, входящие в проект, независимо от того, были в них внесены изменения или нет. Для сборки проекта предназначена команда **Project | Build *<Project1>*** (Проект | Собрать *<Проект1>*).

Запустить проект на выполнение можно как в среде Delphi, так и в среде Windows.

Выполнение проекта *в среде Delphi* осуществляется командой **Run | Run** или нажатием клавиши <F9>. При этом созданное приложение начинает свою работу. Если в файлы проекта вносились изменения, то предварительно выполняется компиляция проекта. Запущенное приложение работает так же, как и запущенное вне среды Delphi, однако имеются некоторые особенности:

- нельзя запустить вторую копию приложения;
- продолжить разработку проекта можно только после завершения работы приложения;
- при зацикливании (зависании) приложения его завершение необходимо выполнять средствами Delphi с помощью команды **Run | Program Reset** (Выполнение | Перезапуск программы) или комбинации клавиш <Ctrl>+<F2>.

Для отладки приложений в среде Delphi можно использовать средства отладчика.

В среде Windows созданное приложение можно запустить, как любое другое приложение, например, с помощью Проводника.

Разработка приложения

Delphi относится к *системам визуального программирования*, называемым также *системами RAD* (Rapid Application Development, быстрая разработка приложений). Разработка приложения в Delphi включает два взаимосвязанных этапа:

- создание пользовательского интерфейса приложения;
- определение функциональности приложения.

Пользовательский интерфейс приложения определяет способ взаимодействия пользователя и приложения, т. е. внешний вид формы (форм) при выполнении приложения и то, каким образом пользователь управляет приложением. Интерфейс конструируется путем размещения в форме компонентов, называемых *интерфейсными компонентами* или *элементами управления*. Создается пользовательский интерфейс приложения с помощью окна Формы, которое в среде разработки представляет собой модель формы времени выполнения.

Функциональность приложения определяется процедурами, которые выполняются при возникновении определенных событий, например, происходящих при действиях пользователя с элементами управления формы.

Таким образом, в процессе разработки приложения в форму помещаются компоненты, для них устанавливаются необходимые свойства и создаются обработчики событий.

Простейшее приложение

Создадим для примера простейшее приложение. Слово "создадим" в данном случае является несколько чрезмерным, т. к. создавать и тем более программировать не придется вообще ничего: Delphi изначально предоставляет готовое приложение, состоящее из одной формы.

Сразу же после создания нового приложения Delphi предлагает разработчику "пустую" форму. Данная форма не является пустой в буквальном смысле слова — она содержит основные элементы окна Windows: заголовок **Form1**, кнопки сворачивания, разворачивания и закрытия окна, изменения размеров окна и кнопку вызова системного меню окна. Именно эта форма отображается при первом запуске Delphi в окне формы.

Любое приложение Windows выполняется в соответствующем окне. Даже если оно ничего не делает в смысле функциональности, т. е. является пустым, то все равно должно иметь свое окно. Delphi — это среда разработки приложений под Windows, поэтому для любого разрабатываемого приложения автоматически предлагается окно (форма), для которой уже созданы два файла — с описанием и модулем.

Итак, простейшее приложение создается автоматически каждый раз в начале работы над новым проектом и является отправной точкой для дальнейших действий. Это приложение имеет минимум того, что нужно любому приложению, выполняемому в среде Windows, и ни одним элементом больше.

Простейшее приложение представляет заготовку или каркас, обеспечивающий разработчика всем необходимым для каждого приложения вообще. Так, не нужно писать свой обработчик событий клавиатуры или драйвер мыши, а также создавать пакет процедур для работы с окнами. Более того, нет необходимости интегрировать драйвер мыши с пакетом для работы с окнами. Это все уже выполнено создателями Delphi, и каркас приложения представляет собой полностью завершенное и функционирующее приложение, которое просто "ничего не делает".

Поясним, что значит фраза "ничего не делает". Окно (а вместе с ним и приложение) действительно ничего не делает с точки зрения пользователя — оно не предоставляет функциональности, специфичной для каждого приложения. Вместе с тем это пустое окно выполняет достаточно большую работу с точки зрения программиста. Например, оно ожидает действий пользователя, связанных с мышью и клавиатурой, и реагирует на изменение своего размера, перемещение, закрытие и некоторые другие команды.

В полной мере оценить эти возможности окна может только программист, который писал приложения под Windows традиционным способом, т. е. вручную и без IDE. Изнутри Windows представляет собой систему с индексами, контекстами, обратными вызовами и множеством других сложнейших элементов, которые надо знать, которыми нужно управлять и в которых очень легко запутаться. Однако поскольку эти элементы имеются в каждом функционирующем приложении Windows, достаточно написать их один раз и в дальнейшем уже пользоваться готовыми блоками. Именно это и осуществляет система Delphi, избавляя тем самым программиста от сложной рутинной работы. О ее объеме можно судить хотя бы по размеру полученного исполняемого файла простейшего приложения — он занимает примерно 360 Кбайт.

Замечание

При компиляции проекта можно использовать специальные пакеты динамически загружаемых библиотек (DLL), что позволяет в значительной степени уменьшить размер приложения. В этом случае для простейшего приложения размер исполняемого файла уменьшается до 15 Кбайт, т. е. более чем в 20 раз. Однако при этом приложение уже не является автономным и в процессе своей работы обращается к пакетам, которые были задействованы при компиляции проекта.

При конструировании приложения разработчик добавляет к простейшему приложению новые формы, управляющие элементы, а также новые обработчики событий.

Создание пользовательского интерфейса приложения

Пользовательский интерфейс приложения составляют компоненты, которые разработчик выбирает в Палитре компонентов и размещает в форме, т. е. компоненты являются своего рода строительными блоками. При конструировании интерфейса приложения действует принцип WYSIWYG (What You See Is What You Get — "что видите, то и получаете"), и разработчик при создании приложения видит форму почти такой же, как и при его выполнении.

Компоненты являются структурными единицами и делятся на визуальные (видимые) и невизуальные (системные). При этом понятия "видимый" и "невидимый" относятся только к этапу выполнения, на этапе проектирования видны все компоненты приложения.

К *визуальным компонентам* относятся, например, кнопки, списки или переключатели, а также собственно форма. Так как с помощью визуальных компонентов пользователь управляет приложением, их также называют *управляющими компонентами* или *элементами управления*. Именно визуальные компоненты образуют пользовательский интерфейс приложения.

К *невизуальным компонентам* относятся компоненты, выполняющие вспомогательные, но не менее важные действия, например, таймер `Timer` или набор данных `Table` (компонент `Timer` позволяет отсчитывать интервалы времени, а компонент `Table` представляет собой записи таблицы базы данных).

При создании интерфейса приложения для каждого компонента выполняются следующие операции:

- выбор компонента в Палитре компонентов и размещение его в форме;
- изменение свойств компонента.

Разработчик выполняет эти операции в окне Формы, используя Палитру компонентов и Инспектор объектов. При этом действия разработчика похожи на работу в среде графического редактора, а сам процесс создания интерфейса приложения больше напоминает конструирование или рисование, чем традиционное программирование. В связи с этим часто работу по созданию интерфейса называют не программированием, а конструированием.

Выбор компонента в Палитре компонентов выполняется щелчком мыши на нужном компоненте, например на кнопке `Button`, в результате чего его значок принимает утопленный (нажатый) вид. Если после этого щелкнуть на свободном месте формы, то на ней появляется выбранный компонент, а его значок в Палитре компонентов принимает обычный (ненажатый) вид. Значки компонентов отражают назначение компонентов, и при наличии небольших практических навыков выбор нужного компонента происходит достаточно быстро. Кроме того, при наведении на каждый компонент указателя мыши отображается всплывающая подсказка с информацией о его назначении.

Обозначения типов объектов в Delphi, в том числе компонентов, начинаются с буквы `T`. Иногда типы (`TButton`, `TLabel`) используются вместо имен (`Button`, `Label`) для обозначения компонентов. Мы в этой книге будем использовать для компонентов имена или типы в зависимости от ситуации.

После размещения компонента в форме система Delphi автоматически вносит изменения в файлы модуля и описания формы. В описание класса формы (файл модуля формы) для каждого нового компонента добавляется строчка формата

```
<Имя компонента>: <Тип компонента>;
```

Имя нового компонента является значением его свойства `Name`, а тип совпадает с типом выбранного в Палитре компонента. Например, для кнопки `Button` эта строчка первоначально будет иметь вид:

```
Button1: TButton;
```

В файле описания для кнопки `Button` может быть записан следующий код:

```
object Button1: TButton
    Left = 88
    Top = 120
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
end
```

Для размещения в форме нескольких одинаковых компонентов удобно перед выбором компонента в Палитре компонентов нажать и удерживать клавишу <Shift>. В этом случае после щелчка мыши в области формы и размещения там выбранного компонента его значок в Палитре остается утопленным, и каждый последующий щелчок в форме приводит к появлению в ней еще одного такого же компонента. Для отмены выбора этого компонента достаточно выбрать другой компонент или щелкнуть мышью на изображении стрелки в левом углу Палитры компонентов.

После размещения компонента в форме можно с помощью мыши изменять его положение и размеры. В случае нескольких компонентов можно выполнять выравнивание или перевод того или иного компонента на передний или задний план. При этом действия разработчика не отличаются от действий в среде обычного графического редактора. Одновременно выделить в форме несколько компонентов можно щелчками мыши на них при нажатой клавише <Shift>.

По умолчанию компоненты выравниваются в форме по линиям сетки, что определяет флажок **Snap to grid** (Выравнивать по сетке), входящий в набор параметров интегрированной среды разработки. В ряде случаев этот флажок приходится отключать, например, при плотном размещении компонентов в форме. По умолчанию шаг сетки равен восьми пикселам, а сетка при проектировании отображается на поверхности формы. Необходимость выравнивания по сетке, видимость сетки (флажок **Display grid** (Отображать сетку)) и размер шага сетки по горизонтали и вертикали устанавливаются на вкладке **Preferences** (Параметры) диалогового окна **Environment Options** (Параметры среды), вызываемого одноименной командой меню **Tools** (Средства).

Внешний вид компонента определяется его свойствами, которые становятся доступными в окне Инспектора объектов, когда компонент выделен в форме и вокруг него появились маркеры выделения (рис. 1.14). Доступ к свойствам самой формы осуществляется аналогично, однако в выбранном состоянии форма не выделяется маркерами. Для выделения (выбора) формы достаточно щелкнуть в любом ее месте, свободном от других компонентов.

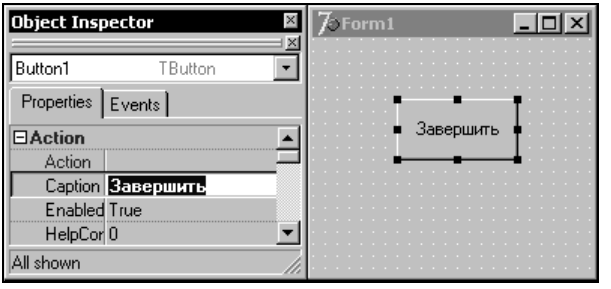


Рис. 1.14. Доступ к свойствам компонента

В раскрывающемся списке, расположенном в верхней части окна Инспектора объектов, отображаются имя компонента и его тип. Выбрать тот или иной компонент и, соответственно, получить доступ к его свойствам можно в этом списке Инспектора объектов. Такой способ выбора удобен в случаях, когда компонент полностью закрыт другими объектами.

В нижней части окна Инспектора объектов слева приводятся имена всех свойств компонента, которые доступны на этапе разработки приложения. Справа для каждого свойства стоит его значение. Отметим, что кроме этих свойств компонент может иметь и свойства, которые доступны только *во время выполнения* приложения.

Свойства представляют собой атрибуты, определяющие способ отображения и функционирования компонентов при выполнении приложения. Первоначально значения свойств задаются по умолчанию. После помещения компонента в форму его свойства можно изменять в процессе проектирования или в ходе выполнения приложения.

Управление свойствами в процессе проектирования заключается в изменении значений свойств компонентов непосредственно в окне формы ("рисование") или с помощью Инспектора объектов.

Разработчик может изменить значение свойства компонента, введя или выбрав нужное значение. При этом одновременно изменяется соответствующий компонент, т. е. уже при проектировании видны результаты сделанных изменений. Например, при изменении заголовка кнопки (свойство `Caption`) оно сразу же отображается на ее поверхности.

Для подтверждения нового значения свойства достаточно нажать клавишу <Enter> или перейти к другому свойству или компоненту. Отмена изменений производится клавишей <Esc>. Если введено недопустимое для свойства значение, то выдается предупреждающее сообщение, а изменение значения отвергается. Изменения свойств автоматически учитываются в файле описания формы, используемом компилятором при создании формы, а при изменении свойства `Name` изменения вносятся и в описание класса формы.

Для большинства свойств компонентов (например, для свойств `Color` (цвет), `Caption` (заголовок) и `Visible` (видимость)) имеются значения по умолчанию.

Для обращения к компоненту в приложении предназначено свойство `Name`, которое образуется автоматически следующим образом: к имени компонента добавляется его номер в порядке помещения в форму. Например, первая кнопка `Button` получает имя `Button1`, вторая — `Button2` и т. д. Первоначально от свойства `Name` получает свое значение и свойство `Caption`.

Обычно разработчик предпочитает давать компонентам более информативные имена, чем имена по умолчанию. При этом целесообразно включать в имя данные о типе компонента и его назначении в приложении. Так, кнопке типа `TButton`, предназначенной для закрытия окна, можно присвоить имя `btnClose` или `ButtonClose`. Каждый разработчик самостоятельно устанавливает удобные правила именования компонентов. В данной книге для простоты мы будем часто использовать имена, назначаемые по умолчанию, например, `Form1`, `Button1` или `Edit1`.

Значения свойств, связанных с размерами и положением компонента (например, `Left` и `Top`), автоматически изменяются при перемещении компонента мышью и настройке его размеров.

Если в форме выделено несколько компонентов, то в окне Инспектора объектов доступны свойства, общие для всех этих компонентов. При этом сделанные в окне Инспектора объектов изменения действуют на все выделенные компоненты.

Для установки значений свойств в Инспекторе объектов используются подключающиеся автоматически редакторы свойств:

- *простой* (текстовый) — значение свойства вводится или редактируется как обычная строка символов, которая интерпретируется как числовой или строковый тип Delphi; используется для таких свойств, как `Caption`, `Left`, `Height` и `Hint`;
- *перечисляемый* — значение свойства выбирается в раскрывающемся списке. Список раскрывается щелчком на стрелке, которая появляется при установке указателя мыши в области значения свойства. Можно не выбирать нужное значение, а ввести его с клавиатуры, однако на практике это обычно не делается, т. к. допускаются только предлагаемые значения. Кроме того, возрастает трудоемкость и увеличивается вероятность ошибки. Используется для таких свойств, как `FormStyle`, `Visible` и `ModalResult`;
- *множественный* — значение свойства представляет собой комбинацию значений из предлагаемого множества. В окне Инспектора объектов слева от имени свойства множественного типа стоит знак "+". Формирование значения свойства выполняется с помощью дополнительного списка, раскрываемого двойным щелчком на имени свойства. Этот список содержит перечень всех допустимых значений свойства, справа от каждого значения можно указать `True` или `False`. Выбор `True` означает, что данное значение включается в комбинацию значений, а `False` — нет. Используется для таких свойств, как `BorderIcons` и `Anchors`;
- *объекта* — свойство является объектом и, в свою очередь, содержит другие свойства (подсвойства), каждое из которых можно редактировать отдельно. Используется для таких свойств, как `Font`, `Items` и `Lines`. В области значения свойства-объекта в

скобках указывается тип объекта, например, (TFont) или (TSrings). Для свойства-объекта слева от имени может стоять знак "+", в этом случае управление его свойствами выполняется так же, как и для свойства множественного типа, т. е. через раскрывающийся список. Этот список в левой части содержит имена подсвойств, а в правой — значения, редактируемые обычным способом. В области значения может отображаться кнопка с тремя точками. Это означает, что для данного свойства имеется специальный редактор, который вызывается нажатием этой кнопки. Так, для свойства Font открывается стандартное окно Windows для установки параметров шрифта.

При выполнении приложения значения свойств компонентов (доступных в окне Инспектора объектов) можно изменять с помощью инструкций присваивания, например, в обработчике события создания формы. Так, изменение заголовка кнопки Button1 можно выполнить следующим образом:

```
Button1.Caption := 'Закрыть';
```

Такой способ требует, однако, большего объема работ, чем в случае использования Инспектора объектов, кроме того, подобные установки вступают в силу только во время выполнения приложения и на этапе разработки не видны, что в ряде случаев затрудняет управление визуальными компонентами. Тем не менее, для наглядности во многих примерах значения отдельных свойств мы будем устанавливать с помощью инструкций присваивания, а не через Инспектор объектов.

Отметим, что есть свойства времени выполнения, недоступные через Инспектор объектов, с которыми можно работать только во время выполнения приложения. К таким свойствам относятся, например, число записей RecordCount набора данных или поверхность рисования Canvas визуального компонента.

Определение функциональности приложения

На любой стадии разработки интерфейсной части приложение можно запустить на выполнение. После компиляции на экране появляется форма приложения, которая выглядит примерно так же, как она была сконструирована на этапе разработки. Форму можно перемещать по экрану, изменять ее размеры, сворачивать и разворачивать, а также закрывать нажатием соответствующей кнопки в заголовке или другим способом. То есть форма ведет себя, как обычное окно Windows.

Реакция на приведенные действия присуща каждой форме и не зависит от назначения приложения и его особенностей. В форме, как правило, размещены компоненты, образующие интерфейс приложения, и задача разработчика — определить для этих компонентов нужную реакцию на те или иные действия пользователя, например, на нажатие кнопки или выбор переключателя. Эта реакция и определяет *функциональность* приложения.

Допустим, что при создании интерфейса приложения в форме была размещена кнопка Button, предназначенная для закрытия окна. По умолчанию эта кнопка получила имя и заголовок Button1, однако заголовок (свойство Caption) посредством окна Инспектора объектов был заменен более осмысленным — Закрыть. При выполнении приложения кнопку Button1 можно нажимать с помощью мыши или клавиатуры. Кнопка отобража-

ет нажатие визуально, однако никаких действий, связанных с закрытием формы, не выполняется. Подобное происходит потому, что для кнопки не определена реакция на какие-либо действия пользователя, в том числе и на ее нажатие.

Чтобы кнопка могла реагировать на то или иное событие, для него необходимо создать или указать процедуру обработки, которая будет вызываться при возникновении этого события. Для создания процедуры обработки события, или обработчика, нужно прежде всего выделить в форме кнопку и перейти на страницу **Events** (События) окна Инспектора объектов, где указываются все возможные для данной кнопки события (рис. 1.15).

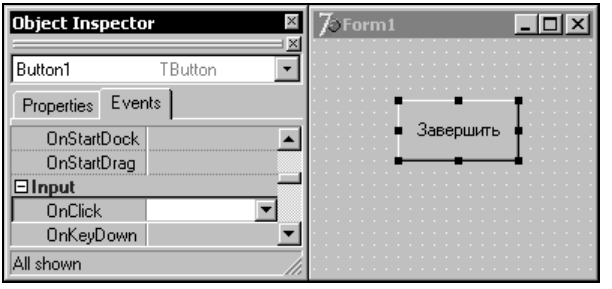


Рис. 1.15. Доступ к событиям компонента

Так как при нажатии кнопки возникает событие `OnClick`, следует создать обработчик именно этого события. Нужно сделать двойной щелчок в области значения события `OnClick`, в результате Delphi автоматически создает в модуле формы заготовку процедуры-обработчика. При этом окно Редактора кода переводится на передний план, а курсор устанавливается в то место процедуры, где программист должен написать код, выполняемый при нажатии кнопки `Button1`. Поскольку кнопка должна закрывать окно, в этом месте можно указать `Form1.Close` или просто `Close`. Код модуля формы будет иметь следующий вид:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Form1.Close;
end;
end.
```

Здесь полужирным начертанием выделен код, набранный программистом; все остальное среда Delphi создала автоматически, в том числе и включение заголовка процедуры-обработчика в описание класса формы `Form1`.

Среда Delphi обеспечивает автоматизацию набора кода при вызове свойств и методов объектов и записи стандартных конструкций языка Delphi. Так, после указания имени объекта и разделяющей точки автоматически появляется список, содержащий доступные свойства и методы этого объекта. При необходимости с помощью комбинации клавиш <Ctrl>+<Пробел> можно обеспечить принудительный вызов этого списка. Имя выбранного свойства или метода автоматически добавляется справа от точки. Если метод содержит параметры, то отображается подсказка, содержащая состав и типы параметров.

Перечень стандартных конструкций языка вызывается нажатием комбинации клавиш <Ctrl>+<J>. После выбора требуемой конструкции автоматически добавляется ее код. Например, при выборе условной инструкции в коде появится следующий текст:

```
if  then
else
```

Имя обработчика `TForm1.Button1Click` образуется прибавлением к имени компонента имени события без префикса `On`. Это имя является значением события, для которого создан обработчик, в нашем случае — для события нажатия кнопки `OnClick`. При изменении через окно Инспектора объектов имени кнопки происходит автоматически переименование этой процедуры во всех файлах (dfm и pas) проекта.

Аналогично создаются обработчики для других событий и других компонентов. Более подробно события рассматриваются при изучении соответствующих компонентов.

Для удаления процедуры-обработчика достаточно удалить код, который программист внес в нее самостоятельно. После этого при сохранении или компиляции модуля обработчик будет удален автоматически из всех файлов проекта.

Замечание

При удалении какого-либо компонента все его непустые обработчики остаются в модуле формы.

Вместо создания нового обработчика для события можно выбрать существующий обработчик, если такой имеется. Для этого нужно в окне Инспектора объектов щелчком на стрелке в области значения свойства раскрыть список процедур, которые можно использовать для обработки этого события. События объекта тоже являются свойствами и имеют определенный для них тип. Для каждого события можно назначить обработчик, принадлежащий к типу этого события. После выбора в списке нужной процедуры она назначается обработчиком события.

Одну и ту же процедуру можно связать с несколькими событиями, в том числе для различных компонентов. Такая процедура называется *общим (разделяемым) обработчиком* и вызывается при возникновении любого из связанных с ней событий. В теле общего обработчика можно предусмотреть действия, позволяющие определить, для какого именно компонента или события вызвана процедура, и в зависимости от этого выполнить нужные команды.

Средства интегрированной среды разработки

Интегрированная среда разработки имеет в своем составе много различных средств, служащих для удобной и эффективной разработки приложений. Отдельные средства, например графический редактор, рассматриваются по мере изложения материала. В этом разделе вы познакомитесь с наиболее общими элементами интегрированной среды разработки Delphi.

Управление параметрами среды

Пользователь может управлять интегрированной средой разработки, настраивая ее отдельные параметры, например, появление окна, отображающего ход компиляции проекта, или автоматическое сохранение редактируемых файлов. Установка параметров выполняется в диалоговом окне **Environment Options** (Параметры среды), вызываемом командой **Tools | Environment Options** (Средства | Параметры среды). Все параметры объединены по группам, размещенным на отдельных страницах (рис. 1.16).

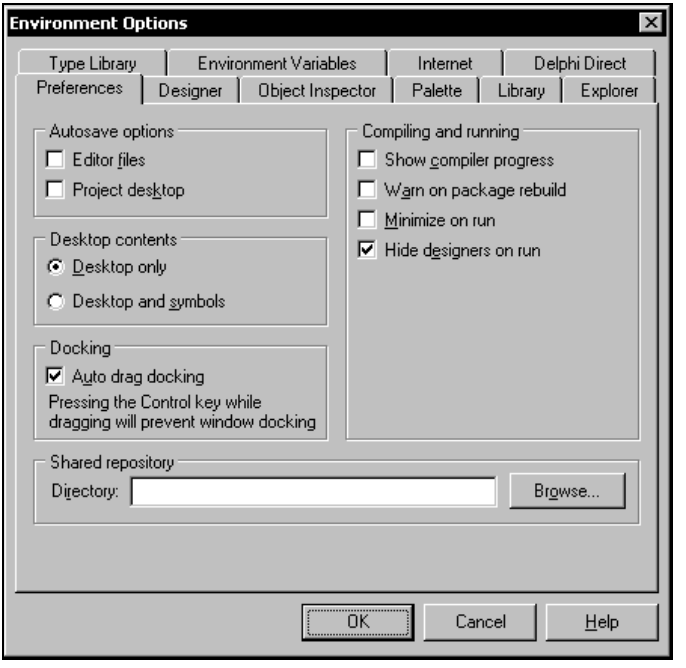


Рис. 1.16. Окно настройки параметров среды разработки

Параметры среды Delphi для каждого проекта сохраняются в файле конфигурации (project configuration file) с расширением cfg.

Менеджер проектов

Менеджер проектов (Project Manager) предназначен для управления проектами и составными частями разрабатываемого приложения. Вызов Менеджера проектов выполняется командой **View | Project Manager** или нажатием комбинации клавиш <Ctrl>+<Alt>+<F11>. Вид окна Менеджера проектов показан на рис. 1.17.

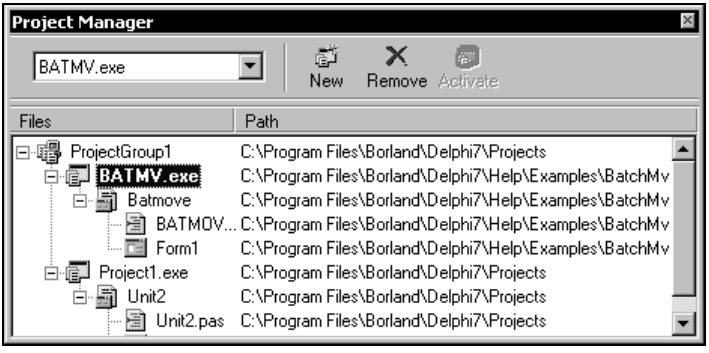


Рис. 1.17. Окно Менеджера проектов

По своей сути система Delphi является SDI-приложением (Single Document Interface, однодокументный интерфейс) и в каждый данный момент времени способна загрузить только один проект, в отличие, скажем, от текстового процессора Microsoft Word. Менеджер проектов частично устраняет это ограничение Delphi и позволяет работать с группой, которая объединяет несколько проектов. В группу удобно объединять проекты, например, при одновременной разработке приложений клиента и сервера или при разработке динамической библиотеки и вызывающего ее приложения. Файл группы проектов имеет расширение bpg и имя `ProjectGroup1` по умолчанию.

Разработчик может:

- добавить в группу новый или уже существующий проект;
- удалить проект группы.

Эти действия выполняются с помощью команд контекстного меню Менеджера проектов или кнопок панели инструментов.

Только один проект в группе является *активным*, т. е. команды компиляции и запуска приложения применяются именно к нему. Активизировать проект можно, выбрав его в раскрывающемся списке под заголовком окна Менеджера проекта, либо командой **Activate** (Активизировать) контекстного меню этого проекта или нажатием одноименной кнопки. Имя активного проекта выделяется полужирным шрифтом.

Встроенный отладчик

Интегрированная среда разработки включает встроенный отладчик приложений, в значительной степени облегчающий поиск и устранение ошибок в приложениях. Средства

отладчика доступны через команды меню **Run** и подменю **View | Debug Windows** (Просмотр | Окна отладки) и позволяют работать в следующих режимах:

- выполнение до указанной инструкции (строки кода);
- пошаговое выполнение приложения;
- выполнение до точки останова (breakpoint);
- включение и выключение точек останова;
- просмотр значений объектов, например, переменных, в окне просмотра;
- установка значений объектов при выполнении приложения.

Установка параметров отладчика выполняется в диалоговом окне **Debugger Options** (Параметры отладчика), вызываемом одноименной командой меню **Tools** (рис. 1.18).

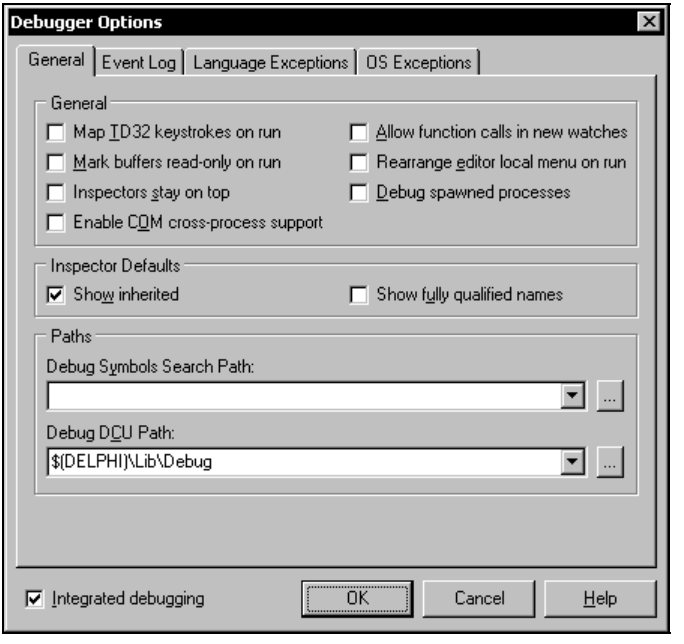


Рис. 1.18. Окно установки параметров отладчика

Включением/выключением отладчика управляет флажок **Integrated debugging** (Интегрированная отладка), который по умолчанию установлен, и отладчик автоматически подключается к каждому приложению. В ряде случаев, например при отладке обработчиков исключений и проверке собственных средств обработки ошибок, этот флажок снимают.

Обозреватель проекта

Обозреватель проекта (Project Browser или Browser) отображает список модулей, классов, типов, свойств, методов и переменных, которые объявлены или использованы в проекте. Обозреватель проекта позволяет просматривать и перемещаться по иерархии классов, модулей и глобальным объектам приложения.

Обозреватель проекта вызывается командой **View | Browser**. Окно Обозревателя проекта **Exploring <...>** (Исследование <...>) разделено на две панели: главную (слева) и детальную (справа). На главной панели в иерархическом виде отображаются доступные объекты выбранного типа, а справа для выбранного объекта детально отображаются его характеристики (рис. 1.19).

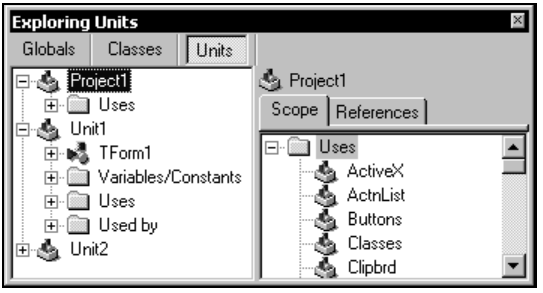


Рис. 1.19. Окно Обозревателя проекта

Для просмотра в окне доступны три типа объектов: **Globals** (Глобальные объекты), **Classes** (Классы) и **Units** (Модули). В зависимости от типа просматриваемого объекта в правой панели окна могут быть использованы следующие варианты детального просмотра характеристик: **Scope** (Область видимости), **Inheritance** (Наследование) и **References** (Ссылки).

Управление отображением данных осуществляется с помощью мыши — путем открытия или закрытия папок в правой и левой панелях Обозревателя. При этом можно отобразить или скрыть соответствующие объекты, например, переменные или константы модуля.

Для управления параметрами отображения объектов, например группировкой по папкам, используется диалоговое окно **Explorer Options** (см. рис. 1.5), открываемое командой **Properties** контекстного меню Обозревателя проекта.

Хранилище объектов

Система Delphi позволяет многократно использовать одни и те же объекты в качестве шаблонов для дальнейшей разработки приложений. Для хранения таких объектов служит специальное Хранилище объектов или Репозиторий (Repository).

Вставить в приложение новый объект можно, открыв командой **File | New | Other** (Файл | Новый | Другой) окно **New Items** (Новые элементы) для выбора нового объекта в Хранилище (рис. 1.20). Это окно можно также открыть нажатием кнопки **New** панели инструментов Менеджера проектов.

В Хранилище находятся самые различные объекты, например, шаблоны приложений, форм, отчетов, а также мастера форм.

Все объекты объединены в следующие группы, размещенные на отдельных страницах:

- **New** — встроенные базовые объекты, используемые при разработке приложений;
- **ActiveX** — объекты COM и OLE, элементы ActiveX, библиотеки ActiveX, активные серверные страницы (ASP);

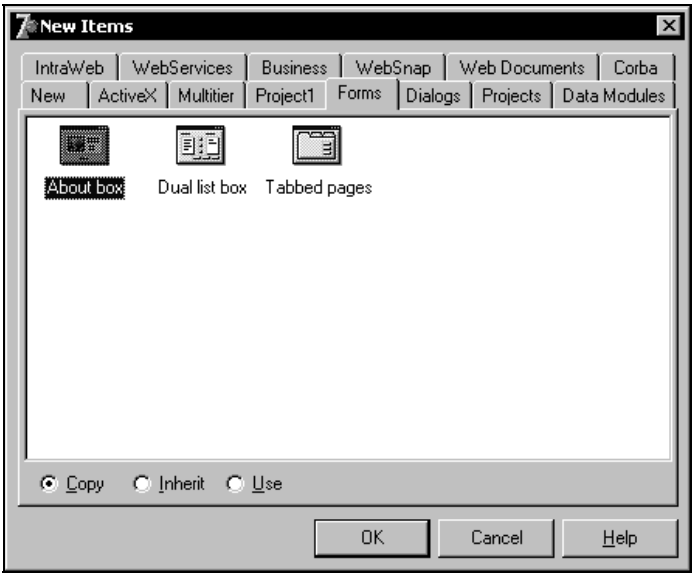


Рис. 1.20. Окно выбора объекта в Хранилище

- **Multitier** — объекты многопоточного приложения (удаленный модуль данных, модуль данных CORBA и др.);
- **Project1** — формы создаваемого приложения;
- **Forms** — формы;
- **Dialogs** — диалоговые окна (стандартное, справочное, для ввода пароля);
- **Projects** — проекты одно- и многодокументного приложений;
- **Data Modules** — модули данных;
- **IntraWeb** — приложения и формы Web;
- **WebServices** — приложение, модуль и интерфейс для SOAP;
- **Business** — мастера форм баз данных и Web-приложений баз данных;
- **WebSnap** — WebSnap-приложения и модули;
- **Web Documents** — Web-документы (HTML, XHTML, WML, XSL);
- **Corba** — CORBA-приложения.

Название страницы **Project1** совпадает с названием создаваемого проекта, а сама страница содержит в качестве шаблонов уже созданные формы приложения (первоначально это одна форма с именем `Form1`). При изменении названия проекта или формы соответственно изменяются их названия в Хранилище объектов. При добавлении к проекту новой формы ее шаблон автоматически добавляется на страницу проекта. В случае удаления из проекта формы ее шаблон также автоматически исключается из Хранилища объектов.

Для добавления нового объекта к проекту необходимо перейти на нужную страницу и указать объект. В примере на рис. 1.20 выбран объект **About box** (Информационное

окно), расположенный на странице **Forms** (Формы). При нажатии кнопки **OK** происходит добавление объекта. Объекты можно добавлять к проекту различными способами, зависящими от выбранного переключателя в нижней части окна выбора нового объекта.

- **Copy** — в проект добавляется копия объекта из Хранилища. В проекте этот объект можно изменять, однако все изменения являются *локальными* в пределах проекта и не затрагивают оригинал, находящийся в Хранилище объектов.
- **Inherit** — от объекта из Хранилища порождается (наследуется) новый объект, который и добавляется к проекту. Разработчик может добавлять к объекту новые компоненты, а также изменять свойства уже существующих элементов, не связанные с их именами. При модификации этого объекта в проекте невозможно удалить какую-либо его составную часть (компонент) или изменить имя (свойство `Name`). По умолчанию подобным образом к проекту добавляются объекты (обычно формы) создаваемого проекта, расположенные на странице **Project1**.

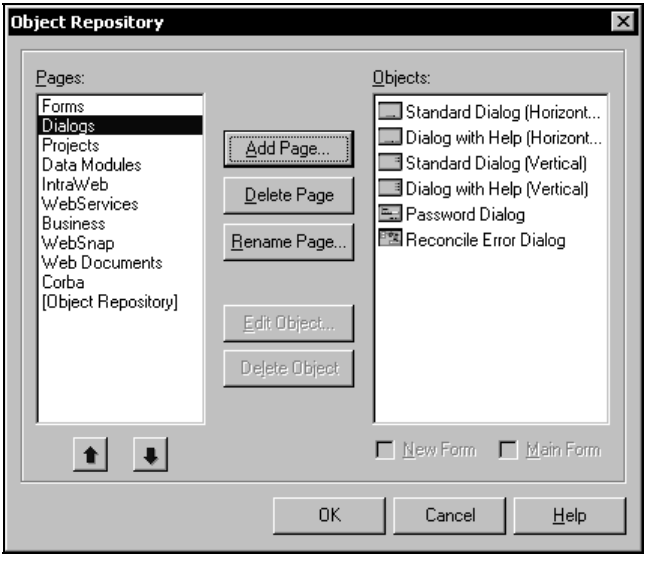


Рис. 1.21. Настройка Хранилища объектов

- **Use** — в проект включается непосредственно сам объект из Хранилища со всеми своими файлами. При изменении в проекте этого объекта изменяется и объект в Хранилище, а также объекты в других проектах, которые таким же образом используют этот объект.

Для настройки Хранилища объектов откройте окно **Object Repository** командой **Tools | Repository** (рис. 1.21).

В процессе настройки в Хранилище объектов можно добавлять (кнопка **Add Page**), удалять из него (кнопка **Delete Page**) и переименовывать страницы (кнопка **Rename Page**), а также редактировать (кнопка **Edit Object**) и удалять (кнопка **Delete Object**) объекты.

Объекты приложения, формы, фрейма, модуля данных и модуля кода тоже можно добавить к проекту через подменю **File | New**, в котором содержатся команды добавления к проекту объектов `Application`, `CLX Application`, `Data Module`, `Form`, `Frame` и `Unit`.

Справочная система

Справочная система Delphi включает в свой состав: стандартную систему справки, справочную помощь через Интернет и контекстно-зависимую справочную помощь.

В стандартной системе справки выделяются две составляющие, вызываемые соответственно командами **Delphi Help** (Помощь Delphi) и **Delphi Tools** (Средства Delphi) меню **Help** (Помощь). При задании любой из команд открывается диалоговое окно с соответствующей справочной информацией, например, **Справка: Delphi Help** (рис. 1.22).

Рис. 1.22. Окно справочной системы Delphi

Вкладка **Содержание** (Contents) окна предоставляет доступ к справочной информации в виде оглавления. Последовательно наводя указатель на элементы оглавления и раскрывая их щелчком мыши, можно переместиться в требуемое место. Щелчок на элементе нижнего уровня иерархии оглавления вызывает отображение соответствующей справочной информации в правой части окна.

Вкладка **Предметный указатель** (Index) диалогового окна справочной системы позволяет выполнить поиск нужной информации с помощью указателя, ключевые слова в котором расположены в алфавитном порядке. Чтобы найти нужную справку, достаточно указать первые несколько букв ключевого слова в поле **1** вкладки или выбрать нуж-

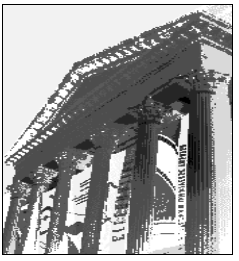
ное слово в списке поля **2** и выполнить на нем щелчок мышью или нажать кнопку **Вывести** (Display). Пользоваться этой вкладкой проще, чем вкладкой **Содержание**, поскольку при поиске справочной информации пользователь не обязан знать, к какому разделу оглавления относится интересующий его вопрос, ему достаточно указать соответствующее ключевое слово.

Вкладка **Поиск** (Find) позволяет выполнить поиск и отображение всех разделов справочной системы, в которых встречается заданная фраза или слово.

Для доступа к справочной системе через Интернет служат команды меню **Help**, которые приводят к запуску Web-обозревателя, например Microsoft Internet Explorer, с открытием соответствующей Web-страницы. Так, команда **Borland Home Page** открывает сайт компании Borland.

Вызов контекстно-зависимой справочной помощи осуществляется нажатием клавиши <F1>, при этом отображаемая справка зависит от того, какой объект (диалоговое окно, пункт меню и т. п.) является активным.

ГЛАВА 2



Язык программирования Delphi

Начиная с версии 7, в среде Delphi для разработки приложений используется язык программирования Delphi, основу которого составляет язык Object Pascal (объектно-ориентированное расширение стандартного языка Pascal). Программирование на языке Delphi подразумевает работу в интегрированной среде разработки приложений (IDE) фирмы Borland. При этом система накладывает ряд ограничений, которые выходят за рамки спецификации языка Object Pascal. В частности, усиливаются соглашения об именовании файлов и программ, которые не обязательно соблюдать за рамками IDE.

Система языка Delphi обеспечивает возможность визуального программирования на нем с помощью библиотеки визуальных компонентов VCL. В среде Delphi 7 можно использовать библиотеку Borland CLX (Component Library for Cross-Platform), представляющую собой межплатформенный вариант библиотеки VCL, который служит для разработки приложений под Windows и Linux.

В данной главе мы рассмотрим основные средства и приемы программирования на языке Delphi.

Основные понятия

В этом разделе рассматриваются алфавит, словарь, структура программы, комментарии, типы данных, инструкции и директивы компилятора, используемые в языке Delphi.

Алфавит

Алфавит языка Delphi включает в себя следующие символы:

- 53 буквы — прописные (`A`...`Z`) и строчные (`a`...`z`) буквы латинского алфавита, а также символ подчеркивания (_);
- 10 цифр (`0`...`9`);
- 23 специальных символа (`+`, `-`, `*`, `/`, `.`, `,`, `:`, `;`, `=`, `>`, `<`, `'`, `(`, `)`, `{`, `}`, `[`, `]`, `#`, `$`, `^`, `@` и пробел).

Комбинации специальных символов образуют следующие *составные* символы:

- := (присваивание);
- <> (не равно);
- .. (диапазон значений);
- <= (меньше или равно);
- >= (больше или равно);
- (* и *) (альтернатива фигурным скобкам { и });
- (. и .) (альтернатива квадратным скобкам [и]).

Словарь языка

Неделимые последовательности знаков алфавита образуют *слова*, отделяемые друг от друга разделителями и несущие определенный смысл в программе. *Разделителями* могут служить пробел, символ конца строки, комментарий, другие специальные символы и их комбинации.

Слова подразделяются на:

- ключевые слова;
- стандартные идентификаторы;
- пользовательские идентификаторы.

Ключевые (зарезервированные) слова являются составной частью языка, имеют фиксированное написание и однозначно определенный смысл, изменить который программист не может. Например, ключевыми являются слова: `Label`, `Unit`, `Goto`, `Begin`, `Interface`. В Редакторе кода ключевые слова выделяются полужирным шрифтом.

Стандартные идентификаторы обозначают заранее определенные разработчиками конструкции языка:

- типы данных;
- константы;
- процедуры и функции.

В отличие от ключевых слов любой из стандартных идентификаторов можно переопределить, но поскольку это может привести к ошибкам, стандартные идентификаторы все же лучше использовать без каких-либо изменений. Примерами стандартных идентификаторов являются слова `Sin`, `Pi`, `Real`.

Пользовательские идентификаторы служат для обозначения имен меток, констант, переменных, процедур, функций и типов данных. Эти имена задаются программистом и должны отвечать следующим правилам:

- идентификатор составляется из букв и цифр;
- идентификатор всегда начинается только с буквы, исключением являются метки, которыми могут быть целые числа без знака в диапазоне 0...9999;
- в идентификаторе можно использовать как строчные, так и прописные буквы, компилятор интерпретирует их одинаково; поскольку использование в идентификато-

рах специальных символов не допускается, для наглядности отдельные составляющие идентификатора полезно выделять прописными буквами, например, `NumberLines` или `btnOpen`;

- между двумя идентификаторами в программе должен быть по крайней мере один разделитель.

Структура программы

Исходный текст программы представляется в виде последовательности строк, при этом текст строки может начинаться с любой позиции. Структурно программа состоит из заголовка и блока.

Заголовок находится в начале программы и имеет вид:

```
Program <Имя программы>;
```

Блок состоит из двух частей: описательной и исполнительной. В *описательной части* содержится описание элементов программы, а в *исполнительной* указываются действия с различными элементами программы, позволяющие получить требуемый результат.

В общем случае описательная часть состоит из следующих разделов:

- подключения модулей;
- объявления меток;
- объявления констант;
- описания типов данных;
- объявления переменных;
- описания процедур и функций.

В конце каждого из указанных разделов ставится точка с запятой.

Замечание

Подчеркнем различие между терминами *"объявление"* и *"описание"*. Суть его заключается в том, что объявление некоторого объекта в программе предполагает выделение основной памяти для его размещения. Описание некоторой конструкции в программе, в отличие от объявления, выделения памяти не требует.

Структуру программы в общем случае можно представить так:

```
Program <Имя программы>;
Uses <Список модулей>;
Label <Список меток>;
Const <Список констант>;
Type <Описание типов>;
Var <Объявление переменных>;
<Описание процедур>;
<Описание функций>;
Begin
  <инструкции>;
End.
```

В структуре конкретной программы любой из разделов описания и объявления может отсутствовать. Разделы описаний и объявлений, кроме раздела подключения модулей, который располагается сразу после заголовка программы, могут встречаться в программе произвольное число раз и следовать в произвольном порядке. При этом все

описания и объявления элементов программы должны быть сделаны до того, как они будут использованы. Рассмотрим подробнее отдельные разделы программы.

Раздел *подключения модулей* состоит из зарезервированного слова `Uses` и списка имен подключаемых стандартных и пользовательских библиотечных модулей. Формат этого раздела:

```
Uses <Имя1>, <Имя2>, ..., <ИмяN>;
```

Например:

```
Uses Crt, Dos, MyLib;
```

Раздел *объявления меток* начинается зарезервированным словом `Label`, за которым следуют имена меток, разделенные запятыми. Формат данного раздела:

```
Label <имя1>, <имя2>, ..., <имяN>;
```

В программе этот раздел может выглядеть так:

```
Label metka1, metka2, 10, 567;
```

В разделе *объявления констант* идентификаторам констант присваиваются их значения. Раздел начинается ключевым словом `Const`, за которым следует ряд конструкций, присваивающих константам значения. Эти конструкции представляют собой имя константы и выражение, значение которого присваивается константе. Имя константы отделено от выражения знаком равенства, в конце конструкции ставится точка с запятой. Формат этого раздела:

```
Const <идентификатор1> = <Выражение1>;
            ...
      <идентификаторN> = <ВыражениеN>;
```

Пример объявления констант:

```
Const st1 = 'WORD'; ch = '5'; n34 = 45.8;
```

Компилятор автоматически распознает тип константы на основании типа выражения.

В Delphi есть много констант, которые можно использовать без предварительного объявления, например, `Nil`, `True` и `MaxInt`.

В разделе *описания типов* описываются пользовательские типы данных. Этот раздел не является обязательным, и типы могут быть описаны неявно в разделе объявления переменных. Раздел описания типов начинается ключевым словом `Type`, за которым располагаются имена типов и их описания, разделенные знаком равенства. В конце описания ставится точка с запятой. Формат раздела:

```
Type <Имя типа1> = <Описание типа1>;
         ...
     <Имя типаN> = <Описание типаN>;
```

Например:

```
Type char2 = ('a'..'z');
     massiv = array[1..100] of real;
     month = 1..12;
```

В Delphi имеется много *стандартных* типов, не требующих предварительного описания: `Real`, `Integer`, `Char`, `Boolean` и др.

Каждая переменная программы должна быть объявлена. Объявление обязательно предшествует использованию переменной. Раздел *объявления переменных* начинается с ключевого слова `Var`, после которого через запятые перечисляются имена переменных и через двоеточие их тип (для каждого типа — свой список переменных, разделитель — точка с запятой). Формат раздела:

```
Var <идентификаторы1> : <тип1>;
       ...
    <идентификаторыN> : <типN>;
```

Например:

```
Var  a, bhg, u7:   real;
     simvol:       char;
     n1,n2:        integer;
```

Замечание

Объявление переменных обеспечивает выделение памяти для размещения переменных в соответствии с их типами, но не присваивание им начальных значений. Программист должен самостоятельно задать нужные начальные значения переменным перед их использованием.

Подпрограммой называют логически законченную и специальным образом оформленную часть программы, которая может вызываться для выполнения из других точек программы неограниченное число раз. В языке Delphi подпрограммы разделяют на два вида: *процедуры* и *функции*. Каждая подпрограмма представляет собой блок и должна быть определена в разделе *описания процедур и функций*. Описание процедур и функций рассматривается далее.

Раздел *инструкций* начинается с ключевого слова `Begin`, после которого следуют инструкции языка, разделенные точкой с запятой. Завершает этот раздел ключевое слово `End`, после которого указывается точка.

Формат раздела:

```
Begin
  <инструкция1>;
  ...
  <инструкцияN>;
End.
```

Здесь могут использоваться любые инструкции языка, например, инструкция присваивания или условная инструкция.

Комментарии

Комментарий представляет собой пояснительный текст, который можно записывать в любом месте программы, где разрешен пробел. Текст комментария ограничен символами `(*` и `*)` (или их эквивалентами `{` и `}`) и может содержать любые символы языка, в том числе русские буквы. Комментарий, ограниченный указанными символами, может занимать несколько строк программы. Однострочный комментарий содержит двойной слэш (`//`) в начале строки.

Примеры комментариев:

```
(* Однострочный комментарий *)
// Другой однострочный комментарий
(* Начало многострочного комментария
          ...
   Окончание многострочного комментария *)
```

Комментарий игнорируется компилятором и не оказывает никакого влияния на выполнение программы. С помощью комментариев можно исключать какие-либо инструкции программы в процессе ее отладки, например, так:

```
sum := 0;
for n := 1 to 100 do begin
       read(x);
//     if x < 0 then x := 0;
       sum := sum + x;
end;
```

Здесь условная инструкция в теле цикла оформлена как комментарий и не будет выполняться.

Типы данных

Обрабатываемые в программе данные подразделяются на переменные, константы и литералы. *Константы* представляют собой данные, значения которых установлены в разделе объявления констант и не изменяются в процессе выполнения программы. *Переменные* объявляются в разделе объявления переменных, однако в отличие от констант получают свои значения уже в процессе выполнения программы, причем допускаются изменения этих значений. К константам и переменным можно обращаться по именам. *Литерал* не имеет идентификатора и представляется в тексте программы непосредственно значением, поэтому литералы также называют просто *значениями*.

Каждый элемент данных принадлежит к определенному типу, при этом тип переменной указывается при ее описании, а тип константы и литерала распознается компилятором автоматически в зависимости от указанного значения.

Тип определяет множество значений, которые могут принимать элементы данных, и совокупность допустимых над ними операций. Например, значения `34` и `67` относятся к целому типу, их, соответственно, можно умножать, складывать, делить и выполнять другие арифметические операции, а значения `'abcd'` и `'sdfh123'` относятся к строковому типу, и их можно соединять (складывать), но нельзя делить или вычитать.

Типы данных можно разделить на следующие группы:

- простые;
- структурные;
- указатели;
- процедурные;
- вариантные.

В свою очередь, простые и структурные типы — это тоже группы, в состав которых входят другие типы, например, целочисленные или массивы. Приводимое деление на

типы в некоторой мере условно — иногда указатели причисляют к простым типам, а строки, которые относятся к структурным типам, выделяют в отдельный тип.

Важное значение имеет понятие *совместимости типов*, которое означает, что типы равны друг другу или один из них может быть автоматически преобразован к другому. Совместимыми, например, являются вещественный и целочисленный типы, т. к. целое число автоматически преобразуется в вещественное (но не наоборот).

Инструкции

Инструкции представляют собой законченные предложения языка, которые выполняют некоторые действия над данными. Инструкции Delphi можно разделить на две группы:

- простые;
- структурированные.

Например, к простым инструкциям относится инструкция присваивания, а к структурированным — условная инструкция и инструкция цикла.

Инструкции разделяются между собой точкой с запятой. Наличие между инструкциями нескольких точек с запятой не является ошибкой, т. к. они обозначают пустые инструкции. Однако имейте в виду, что лишняя точка с запятой в разделе описаний и объявлений уже будет синтаксической ошибкой.

Точка с запятой может не ставиться после слова `begin` и перед словом `end`, т. к. они рассматриваются как операторные скобки, а не как инструкции.

В условных инструкциях и инструкциях выбора точка с запятой (`;`) не ставится после слова `then` и перед словом `else`. Отметим, что в инструкции цикла с параметром наличие точки с запятой сразу после слова `do` синтаксической ошибкой не является, но в этом случае тело цикла будет содержать только пустую инструкцию.

Директивы компилятора

Текст программы может содержать специальные команды, называемые *директивами компилятора*, которые служат для управления режимами компиляции. Директивы компилятора заключаются в фигурные скобки, а в их начале ставится символ `$`. С помощью директив компилятора можно, например, задать способы интерпретации строковых типов, размер стека или подключить файл ресурса.

Программист обычно управляет режимами компиляции с помощью окна параметров проекта (см. рис. 1.13), включая или выключая соответствующие опции на страницах **Compiler** (Компилятор) и **Linker** (Редактор связей). При этом установленные значения параметров сохраняются в файле параметров проекта (dof).

Простые типы данных

Простые типы не содержат в себе других типов, и данные этих типов могут одновременно содержать только одно значение.

К простым относятся следующие типы:

- целочисленные;
- литерные (символьные);
- логические (булевы);
- вещественные.

Все эти типы, кроме вещественного, являются *порядковыми*, т. е. значения каждого из этих типов образуют упорядоченную конечную последовательность. Номера соседних значений в ней отличаются на единицу.

Для значений и имен порядковых типов определены следующие функции:

- `Low(T)` — минимальное значение типа `T`;
- `High(T)` — максимальное значение типа `T`;
- `Ord(X)` — порядковый номер значения выражения `X`;
- `Pred(X)` — значение, предшествующее значению выражения `X`;
- `Succ(X)` — значение, следующее после значения выражения `X`.

Кроме того, к ним применимы следующие процедуры:

- `Dec(X)` — уменьшение значения переменной `X` на единицу;
- `Inc(X)` — увеличение значения переменной `X` на единицу.

Для порядковых типов программист может создавать *перечислимые* и *интервальные* типы. Эти типы также называют *пользовательскими*, или *определяемыми пользователем*. Их применение улучшает наглядность программы и облегчает поиск ошибок.

Некоторые простые типы делятся на физические (фундаментальные) и общие. *Физические типы* закладываются в язык при его разработке и не зависят от особенностей конкретного компьютера. *Общие типы* соответствуют одному из конкретных физических типов, и их использование считается более предпочтительным, т. к. при этом компилятор создает более эффективный код. Названия типов Delphi, независимо от их деления на физические и общие типы, совпадают с названиями типов, многие из которых известны программисту еще по языку Turbo Pascal. Поэтому часто программист использует известные типы данных, например `Real`, `Integer`, `Char` и `Boolean`, не подозревая о подобном делении типов и не зная данной особенности Delphi.

Целочисленные типы

Целочисленные типы, как понятно из их названия, предназначены для целых чисел и могут быть физическими и общими. Физические целочисленные типы Delphi приводятся в табл. 2.1.

Таблица 2.1. *Физические целочисленные типы*

Обозначение	Диапазон	Представление в памяти
`Shortint`	–128 ... 127	1 байт, со знаком
`Smallint`	–32 768 ... 32 767	2 байта, со знаком

Таблица 2.1 *(окончание)*

Обозначение	Диапазон	Представление в памяти
`Longint`	–2 147 483 648 ... 2 147 483 647	4 байта, со знаком
`Int64`	$-2^{63} \ldots 2^{63} - 1$	8 байтов, со знаком
`Byte`	0 ... 255	1 байт, без знака
`Word`	0 ... 65 535	2 байта, без знака
`Longword`	0 ... 4 294 967 295	4 байта, без знака

Кроме физических, определены также два общих типа (табл. 2.2).

Таблица 2.2. *Общие целочисленные типы*

Обозначение	Диапазон	Представление в памяти
`Integer`	–2 147 483 648 ... 2 147 483 647	4 байта, со знаком
`Cardinal`	0 ... 4 294 967 295	4 байта, без знака

Для записи целых чисел можно использовать цифры и знаки "`+`" и "`-`". Если знак числа отсутствует, то число считается положительным. Число может быть представлено как в десятичной, так и в шестнадцатеричной системе счисления. Если число записано в шестнадцатеричной системе, то, чтобы это указать, перед ним ставится знак `$` (без пробела), а допустимый диапазон значений будет `$00000000` ... `$FFFFFFFF`.

Литерные типы

Значениями *литерного типа* являются элементы из набора литер, т. е. отдельные символы. Для символов также имеются физические и общий типы. Физические типы представлены типами `AnsiChar` и `WideChar`.

Символ типа `AnsiChar` занимает один байт, а для кодирования символов используется код ANSI Американского национального института стандартов (American National Standards Institute). Символ типа `WideChar` занимает два байта, а для кодирования символов используется международный набор символов Unicode. Набор символов Unicode включает в свой состав более 60 тысяч элементов и позволяет кодировать, в частности, символы национальных алфавитов. Первые 256 символов Unicode совпадают с кодом ANSI.

Кроме физических типов, в языке Delphi определен один общий тип `Char`, который эквивалентен типу `AnsiChar`.

Для операций с символами имеются следующие функции:

- `Chr(X): Char` — возвращает символ с кодом, равным значению целочисленного выражения `X`;
- `UpCase(C): Char` — преобразует символ `C` к верхнему регистру.

Логические типы

В языке Delphi к логическому относятся следующие типы: `Boolean`, `ByteBool`, `WordBool` и `LongBool`. Из них в программе рекомендуется использовать тип `Boolean`, остальные типы введены в целях совместимости с другими системами программирования. Далее под логическим типом мы будем всегда подразумевать тип `Boolean`.

Этот тип представлен двумя возможными значениями: `True` ("истина") и `False` ("ложь"). Для представления логического значения требуется один байт памяти.

Перечислимые типы

Перечислимый тип задается непосредственно перечислением всех значений (имен), которые может принимать переменная данного типа. Отдельные значения указываются через запятую, а весь список значений заключается в круглые скобки.

Замечание

Значениями перечислимого типа являются сами имена.

Формат описания перечислимого типа:

```
Type <Имя типа> = (<Имя1>, ..., <ИмяN>);
```

Например:

```
Type Day = (Su, Mo, Th, We, To, Fr, St);
...
Var d1,d2,d3: Day;
    Season: (Winter, Spring, Summer, Autumn);
```

Тип `Day` описан явно, и для него определены значения — дни недели. Переменные `d1`, `d2`, `d3` могут принимать одно из перечисленных значений. Попытка присвоить им любое другое значение вызовет программную ошибку. Второй тип определен анонимно (не имеет имени) и задается перечислением значений при объявлении соответствующей переменной `Season`. Последняя оказывается при этом переменной перечислимого типа и может принимать только одно их 4-х указанных в скобках значений (времен года).

Достоинством перечислимых типов является то, что они облегчают контроль значений переменных, т. к. переменной нельзя присвоить не перечисленное предварительно значение. К определенным недостаткам их использования относится то, что при вводе и выводе значений перечислимых типов нельзя указывать имена соответствующих переменных в процедурах ввода/вывода.

Интервальные типы

Интервальные типы описываются путем задания двух констант, определяющих границы допустимых для данных типов значений. Компилятор для каждой операции с переменной интервального типа по возможности проверяет, находится ли значение переменной внутри установленного для нее интервала, и в случае его выхода за границы выдает сообщение об ошибке. На этапе выполнения программы при выходе значе-

ния интервального типа за границы интервала сообщение об ошибке уже не выдается, однако значение переменной будет неверным.

Интервал можно задать только для порядкового типа, т. е. для любого простого типа, кроме вещественного. Обе определяющие интервал константы должны принадлежать одному из простых типов, а значение первой константы должно быть меньше значения второй. Формат описания интервального типа:

```
Type <Имя типа> = <Константа1> .. <Константа2>;
```

Например:

```
Type Day1_31 = 1 .. 31;
...
Var day1, day2 : Day1_31;
```

Переменные `day1` и `day2` имеют тип `Day1_31` и могут принимать значения в диапазоне от `1` до `31`.

Можно определить интервальный тип более универсальным способом, задав границы диапазона не значениями, а именами констант, например, следующим образом:

```
Const min = 1; max = 7;
...
Type NumberWeekDay = min .. max;
...
Var day21, day22 : NumberWeekDay;
```

Здесь переменные `day21` и `day22` имеют тип `NumberWeekDay` и могут принимать значения от `1` до `7`.

Вещественные типы

Переменные *вещественного* типа предназначены для хранения вещественных (действительных) чисел. Вещественные типы представлены в языке Delphi физическими (табл. 2.3) и общим типами.

Таблица 2.3. *Физические типы вещественных чисел*

Обозначение	Минимальное значение	Максимальное значение	Точность (число цифр мантиссы)	Память, байт
`Real`	2.9×10^{-39}	1.7×10^{38}	11—12	6
`Single`	1.7×10^{-45}	3.4×10^{38}	7—8	4
`Double`	5.0×10^{-324}	1.7×10^{308}	15—16	8
`Extended`	3.6×10^{-4951}	1.1×10^{4932}	19—20	10
`Comp`	$-2\times10^{63}+1$	$2\times10^{63}-1$	19—20	8
`Currency`	–922 337 203 685 477.5808	922 337 203 685 477.5807	19—20	8

Общий тип представлен типом `Real`, который соответствует типу `Double`.

Запись вещественных чисел возможна в формах с фиксированной точкой и с плавающей точкой. Вещественные числа с фиксированной точкой записываются по обычным правилам, т. е. целая часть отделяется от дробной десятичной точкой. Перед числом может указываться знак "+" или "-". Если знак отсутствует, то число считается положительным. Для записи вещественных чисел с плавающей точкой указывается порядок числа со знаком, отделенный от мантиссы символом `E` (или `e`). Примеры вещественных чисел: `12.5`, `-137.0`, `+10E+3`.

Типы `Comp` и `Currency` служат для представления вещественных чисел с фиксированной точкой и введены для точных расчетов денежных сумм.

Замечание

Тип `Comp` фактически представляет целые числа, но относится к вещественным типам. При присваивании переменной этого типа вещественного значения оно автоматически округляется до ближайшего целого.

К выражениям вещественных типов применимы следующие функции:

- `Round(X)` — округленное значение выражения `X`;
- `Trunc(X)` — целая часть значения выражения `X`.

Структурные типы данных

Структурные типы объединяют в себе один или несколько других типов, в том числе структурных. К структурным типам относятся:

- строки;
- массивы;
- множества;
- записи;
- файлы;
- классы.

Рассмотрим подробнее перечисленные типы, кроме классов, которые будут изучены позже — после знакомства с подпрограммами и модулями.

Строки

Строки (строковые типы) представлены тремя физическими (табл. 2.4) и одним общим типами.

Таблица 2.4. *Физические строковые типы*

Обозначение	Максимальная длина, символов	Память
`ShortString`	255	2—256 байт
`AnsiString`	Примерно 2×10^{31}	4 байта — 2 Гбайт
`WideString`	Примерно 2×10^{30}	4 байта — 2 Гбайт

Тип `ShortString` представляет собой строку, которая фактически является массивом из 256 элементов — `array[0..255]`. Нулевой байт этого массива (строки) указывает длину

строки. В ранних версиях языка подобная строка обозначалась типом `String`, тип `ShortString` введен в Object Pascal для обеспечения совместимости с этими версиями.

Язык Delphi поддерживает также подтипы типа `ShortString`, максимальная длина которых изменяется в диапазоне от 0 до 255 символов. Они обозначаются целым числом в квадратных скобках, указываемым справа от ключевого слова `string`. Например, инструкции:

```
Var str50: string[50];
```

или

```
type CStr50=string[50];
var str50: CStr50;
```

обеспечивают объявление строковой переменной с именем `str50`, максимальная длина которой составляет 51 символ, т. е. столько, сколько требует описание типа, плюс 1 байт. В случае использования предопределенного типа `ShortString` мы израсходовали бы 256 байт памяти.

Типы `AnsiString` и `WideString` представляют собой динамические массивы, максимальная длина которых фактически ограничена размером основной памяти компьютера. В типе `AnsiString` для символов используется кодировка ANSI, а в типе `WideString` — кодировка Unicode.

Замечание

Операционная система Windows наряду с однобайтовым набором символов обеспечивает поддержку многобайтовых наборов символов, таких как Unicode. При однобайтовом наборе символов каждый байт представляет один символ. В многобайтовом наборе некоторые символы представляются одним байтом, другие символы представляются более чем одним байтом. А именно: первые 128 символов многобайтового набора символов соответствуют карте 7-битовых символов ASCII, и любой из байтов, порядковый номер которого больше 127, является ведущим байтом многобайтового символа.

В наборе символов Unicode каждый символ представляется двумя байтами. Это значит, что в кодировке Unicode строка представляет собой последовательность двухбайтовых слов. Символы и строки Unicode также называют *широкими символами* и *широкими символьными строками*. Первые 256 символов Unicode соответствуют карте набора символов ANSI.

Язык Delphi поддерживает однобайтовые и многобайтовые символы и строки с помощью типов `Char`, `PChar`, `AnsiChar`, `PAnsiChar` и `AnsiString`. В кодировке Unicode символы и строки поддерживаются с помощью типов `WideChar`, `PWideChar` и `WideString`.

Общим типом является тип `String`, который может соответствовать как типу `ShortString`, так и типу `AnsiString`, что определяется директивой компилятора `$H`. По умолчанию используется `{$H+}`, и тип `String` равен типу `AnsiString`. В файле параметров проекта знак "+" директивы (это верно и для других директив) соответствует записи `H=1`, а знак "-" — записи `H=0`. Разработчик обычно управляет интерпретацией типа `String` с помощью окна параметров проекта, устанавливая/снимая флажок **Huge strings** (Большие строки).

Так как строки фактически являются массивами символов, для обращения к отдельному символу строки достаточно указать имя строковой переменной и номер (позицию) этого символа в квадратных скобках, например, `strName[1]`.

Кроме рассмотренных, в языке имеется тип PChar, представляющий так называемую *строку с нулевым окончанием* — в ее конце стоит код #0. Максимальная длина этой строки ограничена размером основной памяти компьютера.

Массивы

Массивом называется упорядоченная индексированная совокупность однотипных элементов, имеющих общее имя. Элементами массива могут быть данные различных типов, включая структурированные. Каждый элемент массива однозначно определяется *именем* массива и *индексом* (номером этого элемента в массиве) или индексами, если массив многомерный. Для обращения к отдельному элементу массива указываются имя этого массива и индекс (индексы) элемента, заключенный в квадратные скобки, например, arr1[3,35], arr1[3][35] или arr3[7].

Количество индексных позиций определяется размерностью массива (одномерный, двумерный и т. д.), при этом размерность не ограничивается. В математике аналогом одномерного массива является вектор, а двумерного — матрица. Индексы элементов массива должны принадлежать порядковому типу. Разные индексы одного и того же массива могут иметь различные типы. Чаще всего индекс имеет целочисленный тип.

Различают массивы статические и динамические. *Статический массив* представляет собой массив, границы индексов и, соответственно, размеры которого задаются при объявлении, т. е. они известны еще до компиляции программы. Формат описания типа статического массива:

```
Array [Имя массива] of <Тип элементов>;
```

Например,

```
Type tm = Array[1 .. 10, 1 .. 100] of real;
...
Var arr1, arr2:   tm;
    arr3:         Array[20 .. 100] of char;
    arr4:         Array['a' .. 'z'] of integer;
```

Переменные arr1 и arr2 являются двумерными массивами по 1000 элементов (10 строк×100 столбцов). Каждый элемент этих массивов представляет собой число типа real. Для объявления массивов arr1 и arr2 введен специальный тип tm. Переменные arr3 и arr4 являются одномерными массивами длиной в 81 символ и 26 целых чисел соответственно.

Динамический массив представляет собой массив, для которого при объявлении указывается только тип его элементов, а размер массива определяется при выполнении программы. Формат описания типа динамического массива:

```
Array of <Тип элементов>;
```

Во время выполнения программы размер динамического массива задается процедурой SetLength(var S; NewLength: Integer), которая для динамического массива S устанавливает новый размер, равный NewLength. Выполнять операции с динамическим массивом и его элементами можно только после задания размеров этого массива.

После задания размера динамического массива для определения его длины, а также минимального и максимального номеров элементов используются функции `Length()`, `Low()` и `High()` соответственно. Нумерация элементов динамического массива начинается с нуля, поэтому функция `Low()` для него всегда возвращает значение `0`.

Приведем пример использования динамического массива.

```
var n: integer;
    m: array of real;
...
SetLength(m, 100);
for n:=0 to 99 do m[n]:=n;
SetLength(m, 200);
```

Здесь после описания динамического массива, содержащего вещественные числа, определяется его размер, равный `100`. Каждому элементу присваивается значение, равное номеру этого элемента в массиве. Так как нумерация элементов массива начинается с нуля, то номер последнего из них равен не `100`, а `99`. После завершения цикла размер массива увеличивается до двухсот.

Для описания типа *многомерного* динамического массива, например двумерного, используется конструкция

```
Array of Array of <Тип элементов>;
```

Для многомерного, в частности двумерного, динамического массива установка новых размеров выполняется для всех индексов с помощью процедуры `SetLength(var S; NewLength1, NewLength2: Integer)`.

Действия над массивом, в том числе и операции ввода/вывода, обычно производятся поэлементно. Поэлементная обработка массивов выполняется, как правило, с помощью циклов. Массив в целом, т. е. как единый объект, может участвовать только в операциях отношения и в инструкции присваивания, причем массивы должны быть полностью идентичными по структуре, т. е. иметь одинаковые типы индексов и одинаковые типы элементов.

Множества

Множество представляет собой совокупность элементов, выбранных из заранее определенного набора значений. Все элементы множества принадлежат одному порядковому типу, число элементов в множестве не может превышать 256. Формат описания множественного типа:

```
Set of <Тип элементов>;
```

Переменная множественного типа может содержать любое количество элементов своего множества — от нуля до максимального. Значения множественного типа заключаются в квадратные скобки. Пустое множество обозначается как `[]`.

Операции, допустимые над множествами, приведены в табл. 2.5.

Таблица 2.5. *Операции над множествами*

Знак операции (оператор)	Операция	Тип результата	Результат
+	Объединение множеств	set of	Неповторяющиеся элементы первого и второго множеств
-	Вычитание множеств	set of	Элементы первого множества, не принадлежащие второму
*	Пересечение множеств	set of	Элементы, общие для обоих множеств
=	Эквивалентность	boolean	True, если множества эквивалентны
<>	Неэквивалентность	boolean	True, если множества не эквивалентны
<=	Проверка вхождения	boolean	True, если первое множество входит во второе
>=	Проверка включения	boolean	True, если первое множество включает второе

Кроме того, есть операция in (проверка членства), которая определяет принадлежность выражения порядкового типа (первого операнда) множеству (второму операнду). Результат операции имеет тип boolean и значение True в случае, если значение принадлежит множеству.

Пример использования множеств:

```
Type MonthDays = set of 1 .. 31;
var Color:  set of (Red, Blue, White, Black);
      Day:  MonthDays;
...
Color := [Blue];
Color := Color - [Blue, Red, White];
Color := Color + [Black];
Color := Color + [Black];
Day := [];
Day := Day - [1];
Day := [2, 4];
Day := Day + [5, 12, 1];
Day := Day - [1];
```

Здесь определяется множественный тип MonthDays и объявляются две переменные Color и Day множественного типа, над которыми выполняются операции объединения и вычитания. В четвертой инструкции присваивания ко множеству Color повторно добавляется значение Black. Такое присваивание корректно, однако значение множества при этом не изменяется. В шестой инструкции присваивания из пустого множества Day вычитается элемент; в результате выполнения этой инструкции значение множества также не изменяется.

В Delphi множественные типы используются, например, для описания типов кнопок в заголовке окна TBorderIcons или типов параметров фильтра TFilterOptions:

```
type TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);
     TBorderIcons = set of TBorderIcon;
type TFilterOption = (foCaseInsensitive, foNoPartialCompare);
     TFilterOptions = set of TFilterOption;
```

Приведенные описания типов содержатся в исходных модулях `Forms` и `Db` соответственно.

Записи

Записи объединяют фиксированное число элементов данных других типов. Отдельные элементы записи имеют имена и называются *полями*. Имя поля должно быть уникальным в пределах записи. Различают фиксированные и вариантные записи. *Фиксированная запись* состоит из конечного числа полей, ее объявление имеет формат:

```
Record
  <Имя поля1> : <Тип поля1>;
  ...
  <Имя поляN> : <Тип поляN>;
end;
```

Вариантная запись, как и фиксированная, имеет конечное число полей, однако предоставляет возможность по-разному интерпретировать области памяти, занимаемые полями. Все варианты записи располагаются в одном месте памяти и позволяют обращаться к ним по различным именам. Отметим, что в данном случае термин "вариантный" не имеет ничего общего с вариантным типом (`Variant`). Формат объявления вариантной записи:

```
Record
  case <Признак> : <Тип признака> of
    <Вариант1> : (<Описание варианта1>);
     ...
    <ВариантN> : (<Описание вариантаN>);
end;
```

Для обращения к конкретному полю необходимо указывать имя записи и имя поля, разделенные точкой, т. е. имя поля является *составным*. С полем можно выполнять все операции, которые допускаются для отдельной переменной этого же типа.

Пример использования записи:

```
var Man: record
           Name:   string;
           Salary: real;
           Note:   string;
           end;
...
Man.Name:='Иванов М.Р.';
Man.Salary:=500;
```

Переменная `Man` является фиксированной записью и имеет поля имени (`Name`), оклада (`Salary`) и примечания (`Note`), каждое своего типа.

Система Delphi предоставляет в распоряжение программиста большое количество уже описанных типов записей, например:

```
type TPoint = record
  X: Longint;
  Y: Longint;
end;
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
```

Здесь тип `TPoint` является фиксированной записью, а тип `TRect` — вариантной записью (рис. 2.1).

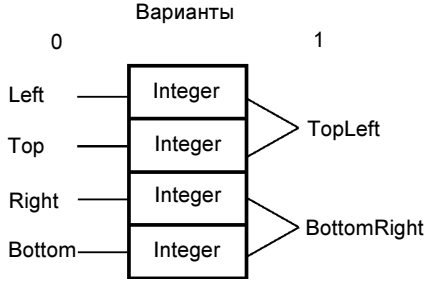


Рис. 2.1. Тип `TRect` — вариантная запись

В варианте `0` запись трактуется как состоящая из четырех отдельных полей типа `Integer`, в варианте `1` — как состоящая из двух полей типа `TPoint`.

Файлы

Файл представляет собой именованную последовательность однотипных элементов, размещенных на внешнем устройстве, чаще всего на диске. Далее мы будем подразумевать, что файл находится именно на диске. Файл имеет много общего с одномерным динамическим массивом, но размещается не в оперативной, а во внешней памяти и не требует предварительного указания размера.

Для выполнения операций с конкретным файлом, размещенным на диске, в программе обычно используется так называемая *файловая переменная*, или *логический файл*. Файловая переменная после описания связывается с некоторым файлом, после чего операции, выполняемые с ней, приводят к соответствующим изменениям в файле. После выполнения всех операций связь между файловой переменной и файлом разрывают, и файловую переменную можно повторно связывать с любым другим файлом этого же типа.

В зависимости от типа элементов различают текстовые, типизированные и нетипизированные файлы. *Текстовый* файл содержит строки символов переменной длины, *ти-*

пизированный файл составляют элементы указанного типа (кроме файлового), а в *нетипизированном* файле находятся элементы, тип которых не указан. Описание файловой переменной, предназначенной для работы с файлом, должно соответствовать типу элементов файла.

Например:

```
Var f1: TextFile;
    f2: File of integer;
    f3: File of real;
    f4: File;
```

Здесь переменная `f1` предназначена для работы с текстовыми файлами. Переменные `f2` и `f3` служат для работы с типизированными файлами, содержащими целые и вещественные числа соответственно. Переменная `f4` предназначена для работы с нетипизированными файлами.

Замечание

В языке Delphi описателем текстовых файлов служит слово `Text`. В системе Delphi многие компоненты имеют одноименное свойство, поэтому при описании текстовых файлов этот описатель в чистом виде использовать нельзя: текстовый файл определяется как `TextFile` или `System.Text` (описатель `Text` из модуля `System`).

Для непосредственной работы с файлами многие компоненты (объекты) предоставляют соответствующие методы, например, `LoadFromFile(const FileName: String)` (загрузить из файла) или `SaveToFile(const FileName: String)` (сохранить в файле). В таких методах файловая переменная не нужна, и в параметре `FileName` указывается просто имя файла.

Подробнее работа с файлами рассматривается в отдельной главе.

Другие типы данных

Указатели

Указатель представляет собой переменную, значением которой является адрес начала размещения некоторых данных в основной памяти. Иными словами, указатель содержит *ссылку* на соответствующий объект. Указатели могут ссылаться на данные любого типа. Переменные типа указателя являются *динамическими* — значения их определяются во время выполнения программы.

Различают указатели *типизированные* и *нетипизированные*. *Типизированный* указатель может ссылаться на данные определенного типа, который указывается при объявлении указателя или при описании типа указателя. При этом используется знак `^`, который ставится перед именем типа адресуемых данных. Формат описания типа для типизированного указателя:

```
Type  <Тип указателя> = ^<Тип адресуемых данных>
```

Нетипизированный указатель имеет тип `Pointer` и может ссылаться на данные любого типа.

Пример объявления переменных-указателей:

```
var p1:  Pointer;
    p2:  ^Integer;
```

Здесь переменная p2 может указывать на данные типа Integer, а переменная p1 — на данные любого типа.

С помощью указателя можно получить доступ к значению адресуемых данных. Для этого служит операция *разыменовывания указателя* — справа от имени указателя приписывается знак ^.

Указателю можно присваивать константу Nil, это означает, что данный указатель ни на что не указывает. Для определения адреса ссылаемого с помощью указателя объекта используется операция @, записываемая перед именем этого объекта.

Например:

```
var  p: ^integer;
  n, k: integer;
...
p := @n;
n := 100;
k := p^ + 10;
```

Первая инструкция присваивает указателю p адрес целочисленной переменной n и позволяет обращаться к ней с помощью конструкции p^. После выполнения всех трех приведенных инструкций присваивания значение переменной k будет равно 110.

В модулях System и SysUtils имеется много типов указателей, например, PString или PVariant, которые можно использовать без их предварительного описания. Модуль System вызывается компилятором автоматически, а имя модуля SysUtils автоматически вносится средой Delphi в список подключаемых модулей раздела Uses для каждой формы.

Процедурные типы

Процедурные типы позволяют интерпретировать процедуры и функции как обычные простые значения, которые, например, можно присваивать переменным или передавать в качестве параметров.

Описание процедурного типа похоже на заголовок процедуры или функции, в котором отсутствует имя подпрограммы. В процедурном типе разрешается использовать методы (подпрограммы, объявленные в классах), для методов при их описании указываются слова of object.

Приведем в качестве примера описание типа TNotifyEvent, который представляет собой тип обработчика многих событий Delphi, например, OnClick или OnChange.

```
type TNotifyEvent = procedure (Sender: TObject) of object;
```

Процедурный тип используется, например, для назначения обработчиков событий, в том числе в окне Инспектора объектов.

Так, в приводимой далее строке кода событию `OnClick` кнопки `Button1` в качестве обработчика назначается процедура `Button1Click`.

```
Button1.OnClick := Button1Click;
```

Эта процедура должна быть предварительно создана и иметь тип, совпадающий с типом `TNotifyEvent` события `OnClick`.

Вариантные типы

Вариантные типы используются для представления значений, которые могут интерпретироваться различными способами. Переменная вариантного типа может содержать значения различных типов и обычно применяется в случаях, когда тип ее значения при компиляции неизвестен или может изменяться в процессе выполнения программы.

Для описания переменной вариантного типа служит ключевое слово `Variant`. Такой переменной можно присваивать значения целочисленных (кроме `Int64`), вещественных, символьных, строковых и логических типов. Все эти типы считаются совместимыми с типом `Variant`, и в случае необходимости компилятор выполняет операцию преобразования типа автоматически.

Пример использования вариантного типа:

```
var v1, v2:  Variant;
         k:  integer;
         x:  real;
...
k := 10;
v1 := k;
x := 23.17;
v2 := x;
v1 := x + 0.5;
```

Здесь вариантной переменной `v1` первоначально присваивается целое значение, а переменной `v2` — вещественное значение. Затем переменной `v1` присваивается вещественное значение.

Для вариантной переменной определены два специфических значения, во многом похожие друг на друга:

- `Unassigned` — назначается переменной при ее описании и указывает, что значение переменной пока не присвоено и не определено;
- `Null` — указывает, что переменная содержит значение неопределенного типа или что значение было потеряно.

Вариантную переменную можно интерпретировать так же, как массив значений (вариантный массив). При этом нельзя назначить обычный статический массив вариантной переменной. Вместо этого следует создать массив `Variant` с помощью одной из двух стандартных функций `VarArrayCreate` или `VarArrayOf`. Например:

```
var X: Variant;
begin
  X := VarArrayCreate([0, 2], varVariant);
```

```
  X[0] := 1;
  X[1] := 'Good luck';
  X[2] := True;
  X[3] := VarArrayOf([1, 10, 50, 200]);
  WriteLn(X[1]);      { Good luck }
  WriteLn(X[3][1]);   { 10 }
end;
```

Первая из инструкций в блоке `begin` создает вариантный массив (длиной `3`) вариантных элементов и назначает его вариантной переменной `X`. Второй параметр в вызове функции `VarArrayCreate` определяет базовый тип элементов массива (в примере вариантный). При вызове стандартной функции `VarArrayOf` создается вариантный массив (из четырех элементов) с набором целочисленных значений, который с помощью инструкции присваивания назначается третьему элементу вариантного массива `X`.

Замечание

Перед использованием результирующие значения элементов массива `Variant` следует самостоятельно преобразовывать к требуемому типу, например, `String`, `Real` или `Integer`.

Для работы с массивом `Variant`, число элементов которого заранее неизвестно, предназначены следующие функции:

- `VarIsArray (const V: Variant): Boolean` — проверяет, является ли параметр `V` массивом типа `Variant`;
- `VarArrayLowBound (const A: Variant; Dim: Integer): Integer` — возвращает нижнюю границу массива, заданного параметром `A`, параметр `Dim` определяет размерность массива;
- `VarArrayHighBound (const A: Variant; Dim: Integer): Integer` — возвращает верхнюю границу массива, заданного параметром `A`, параметр `Dim` определяет размерность массива.

Наряду с типом `Variant` имеется также тип `OleVariant`; главное различие между ними состоит в следующем. Тип `Variant` содержит только доступные для обработки в текущем приложении типы данных. Тип `OleVariant` может содержать типы данных, определенные как совместимые со стандартом OLE Automation, который означает, что типы данных могут передаваться между программами в сети без заботы о том, знают ли на другой стороне, как управлять ими.

Выражения

При выполнении программы осуществляется обработка данных, в ходе которой с помощью выражений вычисляются и используются различные значения. *Выражение* представляет собой конструкцию, определяющую состав данных, операции и порядок выполнения операций над данными. Выражение состоит из:

- операндов;
- знаков операций;
- круглых скобок.

В простейшем случае выражение может состоять из одной переменной или константы. Тип значения выражения определяется типом операндов и составом выполняемых операций.

Операнды представляют собой данные, над которыми выполняются действия. Операндами могут быть константы (литералы), переменные, элементы массивов и обращения к функциям.

Операции определяют действия, которые производятся над операндами. Операции могут быть унарными и бинарными. *Унарная* операция относится к одному операнду, и ее знак записывается перед операндом, например, `-X`. *Бинарная* операция выражает отношение между двумя операндами, и ее знак записывается между операндами, например, `X+Y`.

Круглые скобки используются для изменения порядка выполнения операций.

В зависимости от типов операций и операндов выражения могут быть *арифметическими*, *логическими* и *строковыми*.

Арифметические выражения

Результатом *арифметического выражения* является число, тип которого зависит от типов операндов, составляющих это выражение. В арифметическом выражении можно использовать числовые типы (целочисленные и вещественные), арифметические операции и функции, возвращающие число.

Тип значения арифметического выражения определяется типом операндов и производимыми операциями. Если в операции участвуют целочисленные операнды, то результат операции также будет целочисленного типа. Если хотя бы один из операндов принадлежит к вещественному типу, то результат также будет принадлежать к вещественному типу. Исключением является операция деления, которая всегда приводит к вещественному результату (табл. 2.6).

Таблица 2.6. *Бинарные арифметические операции*

Операция	Назначение	Типы операндов	Тип результата
+	Сложение	Целочисленный Вещественный	Целочисленный Вещественный
−	Вычитание	Целочисленный Вещественный	Целочисленный Вещественный
*	Умножение	Целочисленный Вещественный	Целочисленный Вещественный
/	Деление	Целочисленный Вещественный	Вещественный Вещественный

Унарные арифметические операции + (сохранение знака) и – (отрицание знака) относятся к знаку числа и не изменяют тип числа.

В модулях System, SysUtils и Math содержится много функций для работы с числовыми данными, которые можно использовать в арифметических выражениях. Отметим следующие функции:

- Abs(X) — абсолютное значение X;
- Sqrt(X) — корень квадратный из X;
- Sqr(X) — возведение X в квадрат;
- Ln(X) — натуральный логарифм X;
- Exp(X) — возведение числа *e* в степень X;
- Sin(X) — синус угла X, заданного в радианах.

Кроме функции Sin() есть много других тригонометрических функций, в том числе обратных, например, ArcSin().

В качестве аргумента X функций может указываться число, переменная, константа или выражение.

Например:

```
(x + 12.3) / 30 * sin(2 * alpha)
y + x
exp(3)
```

К целочисленным типам, кроме того, можно применять следующие арифметические операции:

- Div — целочисленное частное от деления двух чисел;
- Mod — целочисленный остаток от деления двух чисел.

Поясним использование арифметических операций для целочисленных операндов на примере.

Пусть переменные a, b и d описаны как целые (integer) и им присвоены значения: a:=10; b:= 7; d:=-56. Тогда в результате выполнения следующих арифметических операций будут получены такие значения:

```
a + 7             17
56 - 8            48
5 * d           -280
56 / b            8.0
56 div b          8
40 div 13         3
40 mod 13         1
```

Для целочисленных типов определяются также следующие побитовые (поразрядные) операции:

- Shl — сдвиг влево;
- Shr — сдвиг вправо;
- And — побитовое умножение;
- Or — побитовое сложение;

- `Xor` — побитовое исключающее сложение;
- `Not` — побитовое отрицание.

Особенностью побитовых операций является то, что они выполняются над операндами поразрядно. Последние четыре операции описаны в табл. 2.7.

Таблица 2.7. *Побитовые операции*

Операция	Описание	Бит 1	Бит 2	Бит результата
`Not`	Отрицание	01		10
`And`	Умножение	0011	0101	0001
`Or`	Сложение	0011	0101	0111
`Xor`	Исключающее сложение	0011	0101	0110

Рассмотрим примеры использования побитовых операций.

Пусть переменные `a` и `b` описаны как целые (`Integer`) и им присвоены значения `a := 186` и `b := 99`. Тогда в результате выполнения следующих битовых операций будут получены значения:

```
Not a        -187
a Or b        251
a And b        34
a Xor b       217
```

Например, результат выполнения операции `And` получен так:

```
10111010         186
01100011          99
-----------
00100010          34
```

Замечание

Кроме побитовых операций `And`, `Or`, `Xor` и `Not` есть одноименные логические операции, применяемые к переменным логического типа.

В языке Delphi отсутствует операция возведения в степень. Возведение числа (выражения) в целую степень можно выполнить в цикле путем многократного умножения числа на себя. Возведение положительного ненулевого числа `X` в произвольную степень `A` производится с помощью выражения `Exp(A*Ln(X))`.

Для целых типов имеется функция `Odd(X): Boolean`, которая анализирует четность выражения `X`.

Логические выражения

Результатом *логического выражения* является логическое значение `True` или `False`. Логические выражения чаще всего используются в условной инструкции и в инструкциях цикла и состоят из:

- логических констант `True` и `False`;
- логических переменных типа `boolean`;

- операций сравнения (отношения);
- логических операций;
- круглых скобок.

Для установления отношения между двумя значениями, заданными выражениями, переменными или константами, используются следующие операции сравнения:

- = (равно);
- < (меньше);
- > (больше);
- <= (меньше или равно);
- >= (больше или равно);
- <> (не равно).

Операции сравнения производятся после вычисления соответствующих выражений. Результатом операции сравнения является значение `False`, если соответствующее отношение не выполняется, и значение `True`, если отношение выполняется.

Замечание

Приоритет операций сравнения меньше, чем приоритет логических операций. Поэтому если содержащее операцию сравнения логическое выражение является операндом логической операции, то его нужно заключить в круглые скобки.

Логические операции (с результатом типа `boolean`) при применении их к логическим выражениям (операндам логического типа) вырабатывают значения также логического типа (табл. 2.8). Логические операции `And`, `Or` и `Xor` являются бинарными, операция `Not` — унарной. Напомним, что в языке Delphi есть одноименные побитовые (поразрядные) операции, выполняющие действия над битами (разрядами) целых чисел *(см. предыдущий раздел)*.

Таблица 2.8. *Логические операции*

Операция	Описание	Операнд 1	Операнд 2	Результат
`Not`	Отрицание	`False`		`True`
		`True`		`False`
`And`	Логическое И	`False`	`False`	`False`
		`False`	`True`	`False`
		`True`	`False`	`False`
		`True`	`True`	`True`
`Or`	Логическое ИЛИ	`False`	`False`	`False`
		`False`	`True`	`True`
		`True`	`False`	`True`
		`True`	`True`	`True`
`Xor`	Исключающее ИЛИ	`False`	`False`	`False`
		`False`	`True`	`True`
		`True`	`False`	`True`
		`True`	`True`	`False`

Примеры логических выражений:

```
x < 10
x+17 >= y
(x > a) and (x < b)
```

Переменные x, a, b и y могут принадлежать, например, к числовым или строковым типам.

Строковые выражения

Результатом строкового выражения является строка символов. Для строк можно применять операцию "+", выполняющую соединение (конкатенацию) двух строк. Определены также следующие функции:

- Length(S): Integer — определение длины строки S;
- Copy(S; Index, Count: Integer): String — выделение из строки S подстроки длиной Count символов; подстрока выделяется, начиная с символа в позиции Index;
- Concat(s1[, s2,..., sn]: String): String — соединение строк s1...sn;
- Pos(Substr: String; S: String): Integer — определение позиции (номера) символа, начиная с которого подстрока Substr входит в строку S, при этом ищется первое вхождение; если подстрока не найдена, то возвращается ноль;

и процедуры:

- Insert(Source: String; var S: String; Index: Integer) — вставка строки Source в строку S, начиная с позиции Index;
- Delete(var S: String; Index, Count: Integer) — удаление из строки S подстроки символов длиной Count, начиная с позиции Index;
- Val(S; var V; var Code: Integer) — преобразование строки S в число V, тип которого зависит от представления числа в строке; параметр Code возвращает код результата операции (ноль, если операция выполнена успешно);
- Str(X [: Width[: Decimals]]; var S) — преобразование значения численного выражения X в строку S.

Кроме перечисленных подпрограмм, большое количество процедур и функций для работы со строками содержится в модуле SysUtils. Назовем следующие функции:

- IntToStr(Value: Integer): String — преобразование значения целочисленного выражения Value в строку;
- StrToInt(const S: String): Integer — преобразование строки S в целое число;
- FloatToStr(Value: Extended): String — преобразование значения вещественного выражения Value в строку;
- StrToFloat(const S: string): Extended — преобразование строки S в вещественное число;
- DateToStr(Date: TDateTime): String — преобразование значения даты в выражении Date в строку;

- `TimeToStr(Time: TDateTime): String` — преобразование значения времени в выражении `Time` в строку;
- `StrToDateTime(const S: String): TDateTime` — преобразование строки `S` в дату и время;
- `StrToDate(const S: String): TDateTime` — преобразование строки `S` в дату;
- `StrToTime(const S: String): TDateTime` — преобразование строки `S` во время;
- `UpperCase(const S: String): String` — перевод символов строки `S` в верхний регистр;
- `LowerCase(const S: String): String` — перевод символов строки `S` в нижний регистр;
- `Trim(const S: String): String` — удаление пробелов и управляющих символов в начале и в конце строки `S`;
- `TrimLeft(const S: String): String` — удаление пробелов и управляющих символов в начале строки `S`;
- `TrimRight(const S: String): String` — удаление пробелов и управляющих символов в конце строки `S`.

Отметим, что для работы с датой и временем используется тип `TDateTime`, а также такие функции, как `Now()`, `Date()` и `Time()`, возвращающие текущие значения даты и времени.

Примеры строковых выражений:

```
'abcdk' + s
'Сумма равна ' + FloatToStr(x)
```

Здесь переменная `s` должна принадлежать к строковому типу, а `x` — к вещественному.

Простые инструкции

Простыми называются инструкции, не содержащие в себе других инструкций. К ним относятся:

- инструкция присваивания;
- инструкция перехода;
- пустая инструкция;
- инструкция вызова процедуры.

Инструкция присваивания

Инструкция присваивания является основной инструкцией языка. Она предписывает вычислить выражение, стоящее справа от знака присваивания, и присвоить результат переменной, имя которой стоит слева от знака присваивания. Переменная и выражение должны иметь совместимые типы, например, вещественный и целочисленный (но не наоборот). Допустимо присваивание значений данных любого типа, кроме файлового. Формат инструкции присваивания:

```
<Имя переменной> := <Выражение>;
```

Вместо имени переменной можно указывать элемент массива или поле записи. Отметим, что знак присваивания := отличается от знака равенства = и имеет другой смысл. Знак присваивания означает, что значение выражения сначала вычисляется, а потом присваивается указанной переменной. Поэтому при условии, что x является числовой переменной и имеет определенное значение, допустимой и правильной будет следующая конструкция:

```
x := x + 1;
```

Примеры инструкций присваивания для переменных со следующими описаниями:

```
var   x, y: real;
         n: integer;
    stroka: string;
...
n := 17 * n — 1;
stroka := 'Дата ' + DateToStr(Date);
x := -12.3 * sin(pi / 4);
y := 23.789E+3;
```

Инструкция перехода

Инструкция перехода предназначена для изменения обычного порядка выполнения инструкций программы. Она используется в случаях, когда после выполнения некоторой инструкции требуется выполнить не следующую по порядку, а какую-либо другую инструкцию. При этом для осуществления перехода инструкция, которой передается управление, должна быть помечена *меткой*. Метка, стоящая перед инструкцией, отделяется от нее двоеточием.

Напомним, что меткой может быть идентификатор или целое число без знака в диапазоне 0...9999, а все метки должны быть предварительно объявлены в разделе объявления меток того блока процедуры, функции или программы, в котором эти метки используются. Формат инструкции перехода:

```
goto <Метка>;
```

Пример использования инструкции перехода:

```
Label m1;
...
goto m1;
...
m1:  <Инструкция>;
```

Передавать управление с помощью инструкции перехода можно инструкциям, расположенным в тексте программы выше или ниже инструкции перехода. Запрещается передавать управление инструкциям, находящимся внутри структурированных инструкций, а также инструкциям, находящимся в других блоках (процедурах, функциях).

Пустая инструкция

Пустая инструкция представляет собой точку с запятой и может быть расположена в любом месте программы, где допускается наличие инструкции. Как и другие инст-

рукции, пустая инструкция может быть помечена меткой. Пустая инструкция не выполняет никаких действий и может быть использована для передачи управления в конец цикла или составной инструкции.

Инструкция вызова процедуры

Инструкция вызова процедуры служит для активизации стандартной или предварительно описанной пользователем процедуры и представляет собой имя этой процедуры со списком передаваемых ей параметров. Более подробно данная инструкция будет рассмотрена при изучении процедур.

Структурированные инструкции

Структурированные инструкции представляют собой конструкции, построенные по определенным правилам из других инструкций. К структурированным инструкциям относятся:

- составная инструкция;
- условная инструкция;
- инструкция выбора;
- инструкции цикла;
- инструкция доступа.

Составная инструкция

Составная инструкция представляет собой группу из произвольного числа любых инструкций, отделенных друг от друга точкой с запятой, и ограниченную операторными скобками `begin` и `end`. Формат составной инструкции:

```
begin <Оператор1>;  ...; <ОператорN>; end;
```

Независимо от числа входящих в нее инструкций составная инструкция воспринимается как единое целое и может располагаться в любом месте программы, где допускается наличие инструкции. Наиболее часто составная инструкция используется в условных инструкциях и инструкциях цикла.

Пример составной инструкции:

```
begin
  Beep;
  Edit1.Text := 'Строка';
  Exit;
end;
```

Приведенная составная инструкция может быть использована в условной инструкции при проверке выполнимости некоторого условия, скажем, для указания действий при возникновении ошибки.

Составные инструкции могут вкладываться друг в друга, при этом глубина вложенности составных инструкций никак не ограничена.

Условная инструкция

Условная инструкция обеспечивает выполнение или невыполнение некоторых инструкций в зависимости от соблюдения определенных *условий*. Условная инструкция в общем случае предназначена для организации ветвления программы на два направления и имеет формат:

```
if <Условие> then <Инструкция1> [else <Инструкция2> ];
```

Условие представляет собой выражение логического типа. Инструкция работает следующим образом: если `Условие` истинно (имеет значение `True`), то выполняется `Инструкция1`, в противном случае — `Инструкция2`. Обе инструкции могут быть составными.

Допускается запись условной инструкции в сокращенной форме, когда слова `else` и `Инструкция2` отсутствуют. В этом случае при невыполнении условия управление сразу передается инструкции, следующей за условной.

Для организации ветвления на три и более направлений можно использовать несколько условных инструкций, вложенных друг в друга. При этом каждое слово `else` соответствует тому `then`, которое непосредственно ему предшествует. Из-за возможной путаницы следует избегать большой глубины вложенности условных инструкций.

Примеры условных инструкций:

```
if x > 0 then x := x + 1 else x := 0;
if q = 0 then a := 1;
```

Инструкция выбора

Инструкция выбора является обобщением условной инструкции и позволяет сделать выбор из произвольного числа имеющихся вариантов, т. е. организовать ветвление на произвольное число направлений. Эта инструкция состоит из выражения, называемого *селектором*, списка вариантов и необязательной ветви `else`, имеющей тот же смысл, что и в условной инструкции. Формат оператора выбора:

```
case <Выражение-селектор> of
       <Список1> : <Инструкция1>;
        ...
       <СписокN> : <ИнструкцияN>
       else <Инструкция>;
end;
```

Выражение-селектор должно быть порядкового типа. Каждый из вариантов выбора (от `Список1` до `СписокN`) представляет собой список констант, отделенных двоеточием от относящейся к данному варианту инструкции, возможно, составной. Список констант выбора состоит из произвольного количества уникальных значений и диапазонов, отделенных друг от друга запятыми. Границы диапазона записываются двумя константами через разделитель "..". Тип констант должен совпадать с типом выражения-селектора.

Инструкция выбора выполняется следующим образом:

1. Вычисляется значение выражения-селектора.
2. Производится последовательный просмотр вариантов на предмет совпадения значения выражения-селектора с константами и значениями из диапазонов соответствующего списка.
3. Если для очередного варианта этот поиск успешный, то выполняется инструкция этого варианта, после чего выполнение инструкции выбора заканчивается.
4. Если все проверки оказались безуспешными, то выполняется инструкция, стоящая после слова `else` (при его наличии).

Пример инструкции выбора:

```
case DayNumber of
  1 .. 5 : strDay:='Рабочий день';
  6, 7   : strDay:='Выходной день'
  else     strDay:='';
end;
```

В зависимости от значения целочисленной переменной `DayNumber`, содержащей номер дня недели, присваивается соответствующее значение строковой переменной `strDay`.

Инструкции цикла

Инструкции цикла служат для организации циклов (повторов). *Цикл* представляет собой последовательность инструкций, которая может выполняться более одного раза. Группу повторяемых инструкций называют *телом цикла*. Для построения цикла в принципе можно применять условную инструкцию и инструкцию перехода, рассмотренные ранее. Однако в большинстве случаев удобно использовать инструкции цикла.

Всего есть три вида инструкций цикла:

- с параметром;
- с постусловием;
- с предусловием.

Обычно если количество повторов известно заранее, то применяется инструкция цикла с параметром, в противном случае — инструкции с пост- или предусловием.

Выполнение инструкции цикла любого вида может быть прервано с помощью инструкции перехода `goto` или предназначенной для этих целей процедуры без параметров `Break`, которая передает управление инструкции, следующей за инструкцией цикла.

С помощью процедуры без параметров `continue` можно задать досрочное завершение очередного повторения тела цикла, что равносильно передаче управления в конец тела цикла.

Инструкции циклов могут быть вложенными друг в друга.

Инструкция цикла с параметром

Инструкция цикла с параметром имеет два возможных формата:

```
for <Параметр> := <Выражение1> to <Выражение2> do <Инструкция>;
```

и

```
for <Параметр> := <Выражение1> downto <Выражение2> do <Инструкция>;
```

Параметр цикла (`Параметр`) представляет собой переменную порядкового типа, которая должна быть определена в том же блоке, где находится инструкция цикла. `Выражение1` и `Выражение2` являются, соответственно, начальным и конечным значениями параметра цикла и должны иметь тип, совместимый с типом параметра цикла.

Инструкция цикла (`Инструкция`) обеспечивает выполнение тела цикла, которым является инструкция после слова `do`, до полного перебора с соответствующим шагом всех значений параметра цикла от начального до конечного. Шаг параметра всегда равен `1` для первого формата цикла и `-1` — для второго, т. е. значение параметра последовательно увеличивается (`for...to`) или уменьшается (`for...downto`) на единицу при каждом повторении цикла.

Цикл может не выполниться ни разу, если для цикла `for...to` значение начального выражения больше конечного, а для цикла `for...downto`, наоборот, значение начального выражения меньше конечного.

Примеры циклов с параметром:

```
var n, k: integer;
...
s := 0;
for n := 1 to 10 do s := s + m[n];
for k := 0 to 2 do
      for n := 5 to 10 do begin
            arr1[k, n] := 0;
            arr2[k, n] := 1;
      end;
```

В первом цикле выполняется расчет суммы десяти значений массива `m`. Во втором случае два цикла вложены один в другой, и в них пересчитываются значения элементов двумерных массивов `arr1` и `arr2`.

Инструкция цикла с постусловием

Инструкцию *цикла с постусловием* целесообразно использовать в случаях, когда тело цикла необходимо выполнить не менее одного раза и заранее неизвестно общее количество повторений цикла. Инструкция цикла с постусловием имеет следующий формат:

```
repeat
  <Инструкция1>;
  ...
  <ИнструкцияN>;
until <Условие>;
```

Условие представляет собой выражение логического типа. Инструкции, заключенные между словами repeat и until, составляют тело цикла и выполняются до тех пор, пока логическое выражение Условие не примет значение True, т. е. тело цикла повторяется при значении Условия, равном False. Так как Условие проверяется только в конце цикла, инструкции тела цикла выполняются минимум один раз.

В теле цикла может находиться произвольное число инструкций без операторных скобок begin и end. По крайней мере одна из инструкций тела цикла должна влиять на значение Условие, в противном случае произойдет зацикливание.

Приведем для иллюстрации цикла с постусловием расчет суммы десяти значений массива m:

```
var x: integer;
  sum: real;
    m: array[1..10] of real;
...
x := 1; sum := 0;
repeat
  sum := sum + m[x];
  x := x + 1;
until(x < 10);
```

Инструкция цикла с предусловием

Инструкцию цикла с предусловием целесообразно использовать в случаях, когда число повторений тела цикла заранее неизвестно и тело цикла может ни разу не выполняться. Эта инструкция аналогична инструкции repeat...until с той лишь разницей, что условие проверяется в начале оператора. Формат инструкции цикла с предусловием:

```
while <Условие> do <Инструкция>;
```

Инструкция тела цикла выполняется до тех пор, пока логическое выражение Условие не примет значение False, т. е. в отличие от цикла с постусловием, данный цикл выполняется при значении логического выражения True.

Снова в качестве примера рассмотрим расчет суммы десяти значений массива m:

```
var x:   integer;
    sum: real;
    m:   array[1..10] of real;
...
x:=1; sum:=0;
while x <= 10 do begin
      sum := sum + m[x];
      x := x + 1;
end;
```

Если перед первым выполнением цикла условие не удовлетворяется (значение логического выражения Условие равно False), то тело цикла не выполнится ни разу, и происходит переход на инструкцию, следующую за инструкцией цикла.

Инструкция доступа

Инструкция доступа служит для удобной и быстрой работы с составными частями объектов, в том числе с полями записей. Напомним, что для обращения к полю записи необходимо указывать имя записи и имя этого поля, разделенные точкой. Аналогичным путем образуется имя составной части какого-либо объекта, например, формы или кнопки. Инструкция доступа имеет следующий основной формат:

```
with <Имя объекта> do <Инструкция>
```

В `Инструкции`, расположенной после слова `do`, для обращения к составной части объекта можно не указывать имя этого объекта, которое уже задано после слова `with`.

Вот пример двух вариантов обращения к составным частям объекта:

```
// Составные имена пишутся полностью
Form1.Canvas.Pen.Color := clRed;
Form1.Canvas.Pen.Width := 5;
Form1.Canvas.Rectangle(10, 10, 100, 100);
```

или

```
// Использование инструкции доступа
with Form1.Canvas do begin
     Pen.Color := clRed;
     Pen.Width := 5;
     Rectangle(10, 10, 100, 100);
end;
```

В обоих вариантах в форме красным карандашом толщиной в пять пикселов рисуется прямоугольник. Для обращения к свойствам и методу (процедуре) поверхности рисования формы удобно использовать инструкцию доступа (второй вариант).

Чтобы не потерять наглядность, мы редко будем использовать в примерах этой книги инструкцию доступа и обычно будем писать составные имена полностью (как в первом варианте приведенного примера).

В инструкции доступа допускается указывать несколько имен объектов.

```
with  <Имя объекта1>,..., <Имя объектаN>  do  <Инструкция>
```

Такой формат эквивалентен следующей конструкции:

```
with <Имя объекта1> do
  with <Имя объекта2> do
     ...
     with <Имя объектаN> do
       <Инструкция>
```

В этом случае для составной части имени объекта, если возможно, применяется `<Имя объектаN>`, иначе — `<Имя объектаN-1>` и т. д. до `<Имя объекта1>`.

Подпрограммы

Подпрограмма представляет собой группу инструкций, логически законченную и специальным образом оформленную. Подпрограмму можно вызывать неограниченное

число раз из различных частей программы. Использование подпрограмм позволяет улучшить структурированность программы и сократить ее размер.

По структуре подпрограмма почти полностью аналогична программе и содержит заголовок и блок, однако в блоке подпрограммы отсутствует раздел подключения модулей. Кроме того, заголовок подпрограммы по своему оформлению отличается от заголовка программы.

Работа с подпрограммой делится на два этапа:

- описание подпрограммы;
- вызов подпрограммы.

Любая подпрограмма должна быть предварительно описана, после чего допускается ее вызов. При описании подпрограммы определяются ее имя, список формальных параметров и выполняемые подпрограммой действия. При вызове указываются имя подпрограммы и список аргументов (фактических параметров), передаваемых подпрограмме для работы.

В различных модулях Delphi есть много стандартных подпрограмм, которые можно вызывать без предварительного описания. Некоторые из них приведены при описании типов данных и выражений. Кроме того, программист может создавать собственные подпрограммы, которые называются *пользовательскими*.

Подпрограммы делятся на *процедуры* и *функции*, которые имеют между собой много общего. Основное различие между ними заключается в том, что функция в качестве результата своей работы может возвращать некоторое значение, присвоенное ее имени, поэтому ее имя можно использовать как операнд выражения.

С подпрограммой взаимодействие осуществляется по управлению и по данным. Взаимодействие *по управлению* заключается в передаче управления из программы в подпрограмму и организации возврата в программу.

Взаимодействие *по данным* заключается в передаче подпрограмме данных, над которыми она выполняет определенные действия. Этот вид взаимодействия может осуществляться следующими основными способами:

- с использованием файлов;
- с помощью глобальных переменных;
- с помощью параметров.

Наиболее часто применяется последний способ. При этом различают параметры и аргументы. *Параметры* (формальные параметры) являются элементами подпрограммы и используются при описании операций, выполняемых подпрограммой.

Аргументы (фактические параметры) являются элементами вызывающей программы. При вызове подпрограммы они замещают формальные параметры. При этом проводится проверка на соответствие типов и количества параметров и аргументов. Имена параметров и аргументов могут различаться, однако их количество и порядок следования должны совпадать, а типы параметров и соответствующих им аргументов должны быть совместимыми.

Для прекращения работы подпрограммы можно использовать процедуру `Exit`, которая прерывает выполнение инструкций подпрограммы и возвращает управление вызывающей программе.

Подпрограммы можно вызывать не только из программы, но и из других подпрограмм.

Процедуры

Описание *процедуры* состоит из заголовка и блока, который, за исключением раздела подключения модулей, не отличается от блока программы. *Заголовок* состоит из ключевого слова `procedure`, имени процедуры и необязательного списка формальных параметров в круглых скобках с указанием типа каждого параметра. Заголовок имеет формат:

```
Procedure <Имя> [ (формальные параметры) ];
```

Для обращения к процедуре используется *инструкция вызова* процедуры. Она состоит из имени процедуры и списка аргументов, заключенного в круглые скобки.

Рассмотрим в качестве примера процедуру обработки события нажатия кнопки `Button1`, в которой, в свою очередь, вызываются две процедуры: `DecodeDate` и `ChangeStr`.

```
procedure TForm1.Button1Click(Sender: TObject);
// Описание пользовательской процедуры ChangeStr
procedure ChangeStr(var Source: string; const char1, char2: char);
label 10;
var n: integer;
begin
10:
  n := pos(char1, Source);
  if n > 0 then begin
     Source[n] := char2;
     goto 10;
   end;
end;
var str1:              string;
    Year, Month, Day:  word;
begin
  // Вызов процедуры DecodeDate
  DecodeDate(Now, Year, Month, Day);
  str1 := Edit1.Text;
  // Вызов пользовательской процедуры ChangeStr
  ChangeStr(str1, '1', '*');
  Edit1.Text := str1;
end;
```

Процедуру `DecodeDate`, которая разбивает дату на отдельные составляющие (год, месяц и день), можно использовать без предварительного описания, т. к. она уже описана в модуле `SysUtils`. Процедура `ChangeStr` выполняет замену в строке `Source` всех вхождений символа, который задает параметр `char1`, на символ, задаваемый параметром `char2`.

Предварительное описание пользовательской процедуры `ChangeStr` выполнено непосредственно в обработчике события нажатия кнопки `Button1`. Если описание вынести

за пределы обработчика, то процедуру ChangeStr можно будет вызывать не только из данного обработчика.

Вызов процедуры ChangeStr обеспечивает замену повсюду в строке str1 символа 1 на символ *.

Функции

Описание *функции* состоит из заголовка и блока. *Заголовок* состоит из ключевого слова Function, имени функции, необязательного списка формальных параметров, заключенных в круглые скобки, и типа возвращаемого функцией значения. Заголовок имеет формат:

```
Function <Имя> [ (Формальные параметры) ] : <Тип результата>;
```

Возвращаемое значение может иметь любой тип, кроме файлового.

Блок функции представляет собой локальный блок, по структуре аналогичный блоку процедуры. В теле функции должна быть хотя бы одна инструкция присваивания, в левой части которой стоит имя функции. Именно она и определяет значение, возвращаемое функцией. Если таких инструкций несколько, то результатом функции будет значение последней выполненной инструкции присваивания. В этих инструкциях вместо имени функции допускается указывать переменную Result, которая создается в качестве синонима для имени функции. В отличие от имени функции, переменную Result можно использовать в выражениях блока функции. С помощью этой переменной можно в любой момент получить внутри блока доступ к текущему значению функции.

Замечание

Имя функции в принципе также можно использовать в выражениях блока функции, однако это приводит к рекурсивному вызову функцией самой себя.

Вызов функции осуществляется по ее имени с указанием в круглых скобках списка аргументов, которого может и не быть. При этом каждому параметру, указанному в заголовке функции, должен соответствовать аргумент того же типа. В отличие от имени процедуры, имя функции может входить в выражения в качестве операнда.

Для примера рассмотрим процедуру обработки события, возникающего при нажатии кнопки Button1. В этой процедуре вызываются две функции: Length и ChangeStr2.

```
procedure TForm1.Button1Click(Sender: TObject);
// Описание функции ChangeStr2
function ChangeStr2(Source: string; const char1, char2: char): string;
label 10;
var n: integer;
begin
  Result := Source;
10:
  n := pos(char1, Result);
  if n > 0 then begin
    Result[n] := char2;
    goto 10;
  end;
end;
```

```
var str1: string;
    n:    integer;
begin
  str1 := Edit1.Text;
  // Вызов функции ChangeStr2
  str1 := ChangeStr2(str1, '1', '*');
  Edit1.Text := str1;
  // Вызов функции Length
  n := Length(str1);
end;
```

Функция `Length` возвращает длину строки и может быть использована без предварительного описания, поскольку оно содержится в модуле `System`. Функция `ChangeStr2` выполняет те же действия, что и процедура `ChangeStr` из примера предыдущего раздела. Вызов функции используется в инструкции присваивания.

Рекурсивные подпрограммы

Иногда требуется, чтобы подпрограмма вызывала сама себя. Такой способ вызова называется *рекурсией*. Рекурсия полезна в случаях, когда основную задачу можно разбить на подзадачи, каждая из которых реализуется по алгоритму, совпадающему с основным.

Рассмотрим, например, вычисление факториала числа:

```
function Fact(n: integer): integer;
begin
  if n <= 0 then Fact := 1 else Fact := n * Fact(n-1);
end;
```

Функция `Fact()` получает число `n` и вычисляет его факториал. При этом, если значение `n` меньше или равно нулю, то функция возвращает значение `1`, в противном случае, уменьшив значение `n` на единицу, функция вызывает сама себя. Подобный рекурсивный вызов выполняется до тех пор, пока `n` не станет равным единице.

Функция `Fact()` и ее параметр имеют тип `Integer`, что соответствует типу `Longint x` и позволяет вычислять факториал для достаточно небольших значений `n` (не более 31). Для увеличения диапазона значений можно использовать вещественные типы, например, `Real`.

Параметры и аргументы

Параметры (формальные параметры) являются элементами подпрограммы и используются при описании производимых в ней действий. Аргументы (фактические параметры) указываются при вызове подпрограммы и замещают параметры при выполнении подпрограммы. Параметры могут иметь любой тип, включая структурированный. Существует несколько видов параметров:

- значение;
- константа;
- переменная;
- нетипизированные константа и переменная.

Параметры, перед которыми в заголовке подпрограммы отсутствуют слова `var` или `const` и за которыми следует их тип, называются *параметрами-значениями*. Например, в следующем описании `a` и `g` являются параметрами-значениями:

```
procedure P1(a: real; g: integer);
```

Параметр-значение обрабатывается как локальная по отношению к подпрограмме переменная. В подпрограмме значения таких параметров можно изменять, однако эти изменения не влияют на значения соответствующих им аргументов, которые при вызове подпрограммы были подставлены вместо формальных параметров.

Параметры, перед которыми в заголовке подпрограммы стоит слово `const` и за которыми следует их тип, называются *параметрами-константами*. Например, в описании

```
procedure P2(const x, y: integer);
```

`x` и `y` являются параметрами-константами. В теле подпрограммы значение параметра-константы изменить нельзя. Параметрами-константами можно оформить те параметры, изменение которых в подпрограмме нежелательно и должно быть запрещено. Кроме того, для параметров-констант строковых и структурных типов компилятор создает более эффективный код.

Параметры, перед которыми в заголовке подпрограммы стоит слово `var` и за которыми следует их тип, называются *параметрами-переменными*. Например, в следующем описании `d` и `f` являются параметрами-переменными:

```
function F1(var d, f: real; q17: integer): real;
```

Параметр-переменная используется в случаях, когда значение должно быть передано из подпрограммы в вызывающий блок. В этом случае при вызове подпрограммы параметр замещается аргументом-переменной, и любые изменения формального параметра отражаются на аргументе. Таким способом можно вернуть результаты из подпрограммы по окончании ее работы.

Для параметров-констант и параметров-переменных допускается не указывать их тип, т. е. считать их *нетипизированными*. В этом случае подставляемые на их место аргументы могут быть любого типа, и программист должен самостоятельно интерпретировать типы параметров в теле подпрограммы. Примером задания нетипизированных параметров-констант и параметров-переменных является следующее описание:

```
function F2 (var a1; const t2): integer;
```

Замечание

Группы параметров в описании подпрограммы разделяются точкой с запятой.

Модули

Кроме программ, структуру которых мы только что рассмотрели, средства языка позволяют создавать модули. В отличие от программы, *модуль* не может быть автономно запущен на выполнение и содержит элементы, например, переменные и подпрограммы, которые допускается использовать в программе или в других модулях. Для того чтобы можно было использовать средства модуля, его необходимо подключить, указав

имя этого модуля в разделе `uses`. Типичными примерами модулей являются `System` и `SysUtils`, содержащие большое количество стандартных подпрограмм (некоторые из них уже были рассмотрены). Напомним, что для каждой формы приложения создается отдельный модуль.

Компилятор распознает модуль по его заголовку и создает в результате своей работы не исполняемый файл (exe), как это было для приложения, а файл модуля с расширением dcu.

Модуль состоит из заголовка, в котором после ключевого слова `unit` указывается имя модуля, и четырех разделов: интерфейса (`Interface`), реализации (`Implementation`), инициализации (`Initialization`) и деинициализации (`Finalization`).

Модуль имеет следующую структуру:

```
Unit <Имя модуля>;
  // Раздел интерфейса
  Interface
  Uses <Список модулей>;
  Const <Список констант>;
  Type <Описание типов>;
  Var <Объявление переменных>;
  <Заголовки процедур>;
  <Заголовки функций>;

  // Раздел реализации
  Implementation
  Uses <Список модулей>;
  Const <Список констант>;
  Type <Описание типов>;
  Var <Объявление переменных>;
  <Описание процедур>;
  <Описание функций>;

  // Раздел инициализации
  Initialization
  <Инструкции>

  // Раздел деинициализации
  Finalization
  <Инструкции>
end.
```

В разделе *интерфейса* размещаются описания идентификаторов, которые должны быть доступны всем модулям и программам, использующим этот модуль и содержащим его имя в списке `uses`. В разделе интерфейса объявляются типы, константы, переменные и подпрограммы. При этом для подпрограмм указываются только их заголовки. Другие используемые модули указываются в списке `uses`. Раздел интерфейса начинается ключевым словом `Interface`.

В разделе *реализации* располагается код подпрограмм, заголовки которых были приведены в разделе интерфейса. Порядок следования подпрограмм может не совпадать

с порядком расположения их заголовков, приводимых в разделе интерфейса. Кроме того, допускается оставлять в заголовке только имя подпрограммы, т. к. список параметров и тип результата функции уже были предварительно указаны. В разделе реализации можно также описывать типы, объявлять константы и переменные и описывать подпрограммы, которые используются только в этом модуле и за его пределами не видны. Раздел интерфейса начинается словом `Implementation`.

В разделе *инициализации* располагаются инструкции, выполняемые в начале работы программы, которая подключает данный модуль. Разделы инициализации модулей выполняются в том порядке, в котором они перечислены в списке раздела `uses` программы. Раздел инициализации начинается словом `Initialization` и является необязательным.

При наличии раздела инициализации в модуле можно использовать раздел *деинициализации*, который начинается словом `Finalization` и является необязательным. В этом разделе располагаются инструкции, выполняемые при завершении программы. Разделы деинициализации модулей выполняются в порядке, обратном порядку их перечисления в списке `uses` программы.

Особенности объектно-ориентированного программирования

Язык Delphi реализует концепцию объектно-ориентированного программирования (ООП). Это означает, что функциональность приложения определяется набором взаимосвязанных задач, каждая из которых становится самостоятельным объектом. У объекта есть свойства (т. е. характеристики, или атрибуты), методы, определяющие его поведение, и события, на которые он реагирует. Одним из наиболее важных понятий ООП является класс. *Класс* представляет собой дальнейшее развитие концепции типа и объединяет в себе задание не только структуры и размера переменных, но и выполняемых над ними операций. *Объекты* в программе всегда являются экземплярами того или иного класса (аналогично переменным определенного типа).

Основные концепции ООП

К основным концепциям ООП относятся следующие:

- инкапсуляция;
- наследование;
- полиморфизм.

Инкапсуляция представляет собой объединение данных и обрабатывающих их методов (подпрограмм) внутри класса (объекта). Это означает, что в классе инкапсулируются (объединяются и помещаются внутрь) поля, свойства и методы. При этом класс получает определенную функциональность, например, обеспечивая полный набор средств для создания программы поддержки некоторого элемента интерфейса (окна Windows, редактора и т. п.) или прикладной обработки.

Наследование — это процесс порождения новых объектов-потомков от существующих объектов-родителей, при этом потомок наследует от родителя все его поля, свойства и

методы. В дальнейшем наследуемые поля, свойства и методы можно использовать в неизменном виде или переопределять (модифицировать).

Просто наследование большого смысла не имеет, поэтому в объект-потомок добавляются новые элементы, определяющие его особенность и функциональность. Удалить какие-либо элементы родителя в потомке нельзя. В свою очередь, от нового объекта можно породить следующий объект, в результате образуется дерево объектов (называемое также *иерархией классов*).

В корне этого дерева находится базовый класс `TObject`, который реализует наиболее общие для всех классов элементы, например, действия по созданию и удалению объекта. Чем дальше тот или иной класс отстоит в дереве от базового класса, тем большей специфичностью он обладает.

Пример объявления нового класса:

```
TAnyClass = class (TParentClass)
  // Добавление к классу TParentClass новых
  // и переопределение существующих элементов
  ...
end;
```

Сущность *полиморфизма* заключается в том, что методы различных классов могут иметь одинаковые имена, но различное содержание. Это достигается переопределением родительского метода в классе-потомке. В результате родитель и потомок ведут себя по-разному. При этом обращение к одноименным методам различных объектов выполняется аналогично.

Классы и объекты

В языке Object Pascal классы — это специальные типы данных, используемые для описания объектов. Соответственно *объект*, имеющий тип какого-либо класса, является *экземпляром* (instance) этого класса или переменной этого типа.

Класс представляет собой особый тип записи, имеющий в своем составе такие элементы (члены, member), как поля, свойства и методы. *Поля класса* аналогичны полям записи и служат для хранения информации об объекте. *Методами* называются процедуры и функции, предназначенные для обработки полей. *Свойства* занимают промежуточное положение между полями и методами.

С одной стороны, свойства можно использовать как поля, например, присваивая им значения с помощью инструкции присваивания; с другой стороны, внутри класса доступ к значениям свойств выполняют методы класса.

Описание класса имеет следующую структуру:

```
type <Имя класса> = class(<Имя класса-родителя>)
  private
     <Частные описания>;
  protected
     <Защищенные описания>;
  public
     <Общедоступные описания>;
```

```
  published
     <Опубликованные описания>;
end;
```

В приведенной структуре описаниями являются объявления свойств, методов и событий.

Пример описания класса:

```
type
   TColorCircle = class(TCircle);
      FLeft,
      FTop,
      FRight,
      FBottom: Integer;
      Color: TColor;
  end;
```

Здесь класс `TColorCircle` создается на основе родительского класса `TCircle`. По сравнению с родительским, новый класс дополнительно содержит четыре поля типа `Integer` и одно поле типа `TColor`.

Если в качестве родительского используется класс `Tobject`, который является базовым классом для всех классов, то его имя после слова `class` можно не указывать. Тогда первая строка описания будет выглядеть так:

```
type TNewClass = class
```

Для различных элементов класса можно устанавливать разные права доступа (видимости), для чего в описании класса используются отдельные разделы, обозначенные специальными спецификаторами видимости.

Разделы `private` и `protected` содержат *защищенные* описания, которые доступны внутри модуля, в котором они находятся. Описания из раздела `protected`, кроме того, доступны для порожденных классов за пределами названного модуля.

Раздел `public` содержит *общедоступные* описания, которые видимы в любом месте программы, где доступен сам класс.

Раздел `published` содержит *опубликованные* описания, которые в дополнение к общедоступным описаниям порождают динамическую (т. е. во время выполнения программы) информацию о типе (Run-Time Type Information, RTTI). По этой информации при выполнении приложения производится проверка на принадлежность элементов объекта тому или иному классу. Одним из назначений раздела `published` является обеспечение доступа к свойствам объектов при конструировании приложений. В Инспекторе объектов видны те свойства, которые являются опубликованными. Если спецификатор `published` не указан, то он подразумевается по умолчанию, поэтому любые описания, расположенные за строкой с указанием имени класса, считаются опубликованными.

Объекты как экземпляры класса объявляются в программе в разделе `var` как обычные переменные. Например:

```
var
  CCircle1: TColorCircle;
  CircleA:  TCircle;
```

Как и в случае записей, для обращения к конкретному элементу объекта (полю, свойству или методу) указывается имя объекта и имя элемента, разделенные точкой, т. е. имя элемента является *составным*.

Пример обращения к полям объекта:

```
var
  CCircle1: TColorCircle;
begin
  ...
  CCircle1.FLeft:=5;
  CCircle1.FTop:=1;
  ...
end;
```

Здесь приведено непосредственное обращение к полям объекта, обычно это делается с помощью методов и свойств класса.

Поля

Поле класса представляет собой данные, содержащиеся в классе. Поле описывается как обычная переменная и может принадлежать к любому типу.

Пример описания полей:

```
type  TNewClass = class(TObject)
  private
     FCode: integer;
     FSign: char:
     FNote: string;
end;
```

Здесь новый класс `TNewClass` создается на основе базового класса `Tobject` и получает в дополнение три новых поля: `FCode`, `FSign` и `FNote`, имеющих, соответственно, целочисленный, символьный и строковый типы. Согласно принятому соглашению имена полей должны начинаться с префикса `F` (от англ. *Field* — поле).

При создании новых классов класс-потомок наследует все поля родителя, при этом удалить или переопределить эти поля нельзя, но можно добавить новые. Таким образом, чем дальше по иерархии какой-либо класс находится от родительского класса, тем больше полей он имеет.

Напомним, что изменение значений полей обычно выполняется с помощью методов и свойств объекта.

Свойства

Свойства (property) реализуют механизм доступа к полям. Каждому свойству соответствуют поле, содержащее значение свойства, и два метода, обеспечивающих доступ к этому полю. Описание свойства начинается со слова `property`, при этом типы свойства и соответствующего поля должны совпадать. Ключевые слова `read` и `write` являются зарезервированными внутри объявления свойства и служат для указания методов клас-

са, с помощью которых выполняется чтение значения поля, связанного со свойством, или запись нового значения в это поле.

Пример описания свойств:

```
type  TNewClass = class(TObject)
  private
     FCode: integer;
     FSign: char:
     FNote: string;
  published
     property Code: integer read FCode write FCode;
     property Sign: char read FSign write FSign;
     property Note: string read FNote write FNote;
  end;
```

Для доступа к полям `FCode`, `FSign` и `FNote`, которые описаны в защищенном разделе и недоступны для других классов, используются свойства `Code`, `Sign` и `Note` соответственно.

Методы

Метод представляет собой подпрограмму (процедуру или функцию), являющуюся элементом класса. *Описание метода* похоже на описание обычной подпрограммы модуля. Заголовок метода располагается в описании класса, а сам код метода находится в разделе реализации. Имя метода в разделе реализации является составным и включает в себя тип класса.

Например, описание метода `Button1Click` будет выглядеть так:

```
interface
...
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;
...
implementation
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  Close;
end;
```

Метод, объявленный в классе, может вызываться различными способами, что зависит от вида этого метода. Вид метода определяется модификатором, который указывается в описании класса после заголовка метода и отделяется от заголовка точкой с запятой. Приведем некоторые модификаторы:

- `virtual` — виртуальный метод;
- `dynamic` — динамический метод;

- `override` — переопределяемый метод;
- `message` — обработка сообщения;
- `abstract` — абстрактный метод.

По умолчанию все методы, объявленные в классе, являются *статическими* и вызываются как обычные подпрограммы.

Методы, которые предназначены для создания или удаления объектов, называются *конструкторами* и *деструкторами* соответственно. Описания данных методов отличаются от описания обычных процедур только тем, что в их заголовках стоят ключевые слова `constructor` и `destructor`. В качестве имен конструкторов и деструкторов в базовом классе `Tobject` и многих других классах используются имена `Create` и `Destroy`.

Прежде чем обращаться к элементам объекта, его нужно создать с помощью конструктора. Например:

```
ObjectA := TOwnClass.Create;
```

Конструктор выделяет память для нового объекта в "куче" (heap), задает нулевые значения для порядковых полей, значение `nil` — для указателей и полей-классов, строковые поля устанавливает пустыми, а также возвращает указатель на созданный объект.

При выполнении конструктора часто также осуществляется инициализация элементов объекта с помощью значений, передаваемых в качестве параметров конструктора.

Приведем примеры использования конструктора и деструктора:

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    procedure PenChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ...
  end;
// Описание конструктора Create класса TShape
constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner);   // Инициализация унаследованных частей
  Width  := 65;              // Изменение унаследованных свойств
  Height := 65;
  FPen   := TPen.Create;     // Инициализация новых полей
  FPen.OnChange := PenChanged;
end;
```

В конструкторе класса-потомка сначала вызывается конструктор родителя, а затем выполняются остальные действия. В классе-потомке директива `override` (переопределить) обеспечивает возможность родительскому классу использовать новый метод. Ключевое слово `inherited` служит для вызова методов родительского класса.

Описание методов, а также полей и свойств класса будет продолжено при рассмотрении вопросов, связанных с созданием компонентов.

Сообщения и события

В основе операционной системы Windows лежит использование механизма сообщений, которые "документируют" все производимые действия, например, нажатие клавиши, передвижение мыши или тиканье таймера. Приложение получает сообщение в виде записи заданного типа, определяемого как:

```
type
  PMsg = ^TMsg;
  Msg = packed record
    hwnd:    HWND;
    message: UINT;
    wParam:  WPARAM;
    lParam:  LPARAM;
    time:    DWORD;
    pt:      TPoint
end;
```

Поля этой записи содержат следующую информацию:

- `hwnd` — дескриптор управляющего элемента, которому предназначено сообщение;
- `message` — код сообщения;
- `wParam` и `lParam` — дополнительная информация о сообщении;
- `time` — время обработки сообщения Windows;
- `pt` — координаты указателя мыши во время генерации сообщения.

Система Delphi преобразует сообщение в свой формат, для которого используется запись следующего типа:

```
PMessage = ^TMessage;
TMessage = record
    Msg: Cardinal;
    case Integer of
      0: (
        WParam:   Longint;
        LParam:   Longint;
        Result:   Longint);
      1: (
        WParamLo: Word;
        WParamHi: Word;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
end;
```

Типы `Msg`, `TMessage`, а также константы, используемые при посылке сообщений, описаны в файлах windows.pas и message.pas.

Для обработки сообщений, посылаемых ядром Windows и различными приложениями, используются специальные методы, описываемые с помощью модификатора `message`, после которого указывается идентификатор сообщения.

Метод обработки сообщения обязательно должен быть процедурой, имеющей один параметр, который при вызове метода содержит информацию о поступившем сообщении. Имя метода программист выбирает самостоятельно, для компилятора оно не имеет значения, т. к. данный метод является *динамическим* и его вызов выполняется по таблице динамических методов.

Метод может полностью или частично перекрывать метод-предок, который обрабатывает это сообщение. Если метод только модифицирует метод-предок, то для вызова последнего используется метод `Inherited`. При этом не нужно указывать имя метода-предка и его параметры, т. к. вызов будет выполнен автоматически.

Рассмотрим в качестве примера обработку сообщения Windows, посылаемого при изменении размеров окна.

```
type
  TForm1 = class(TForm)
  // Объявление метода обработки сообщения
  procedure MyPaint(Var Param); message WM_Size;
end;
...
// Код метода обработки сообщения
procedure TForm1.MyPaint(Var Param);
begin
  // Вызов метода-предка
  inherited;
  // Очистка поверхности формы
  Form1.Refresh;
  // Вывод красной рамки
  Form1.Canvas.Pen.Color:=clRed;
  Form1.Canvas.Brush.Style:=bsClear;
  Form1.Canvas.Rectangle(0, 0, Form1.ClientWidth, Form1.ClientHeight);
end;
```

По периметру формы выводится красная рамка с помощью процедуры `MyPaint`, которая является обработчиком сообщения `WM_Size`. Это сообщение посылается при изменении размеров окна. В данном примере рамка перерисовывается (вместе с формой) только при изменении размеров окна, но не при его перекрытии другими окнами, т. к. в этом случае посылается сообщение `WM_Paint`, которое здесь не анализируется. Параметр `Param` процедуры нигде не используется, однако должен быть указан в заголовке процедуры.

Обычно в Delphi не требуется обязательная обработка непосредственных сообщений Windows, т. к. в распоряжение программиста предоставляются события, работать с которыми намного проще и удобнее. *Событие* представляет собой свойство процедурного типа, предназначенное для обеспечения реакции на те или иные действия. Присваивание значения этому свойству (событию) означает указание метода, вызываемого при наступлении события. Соответствующие методы называются *обработчиками событий*.

Пример назначения обработчика события:

```
Application.OnIdle:=IdleWork;
```

В качестве обработчика события OnIdle, возникающего при простое приложения, объекту приложения назначается процедура IdleWork. Поскольку объект Application доступен только при выполнении приложения, такое присваивание нельзя выполнить через Инспектор объектов.

События Delphi имеют различные типы, зависящие от вида этого события. Самым простым является тип TNotifyEvent, характерный для нотификационных (уведомляющих) событий. Этот тип описан следующим образом:

```
type TNotifyEvent = procedure(Sender: TObject) of object;
```

и содержит один параметр Sender, указывающий объект-источник события. Многие события более сложного типа, наряду с другими параметрами, также имеют параметр Sender.

Так как события являются свойствами, их значения можно изменять в процессе выполнения приложения, т. е. можно *динамически* изменять реакцию объекта на одно и то же событие. При этом допускается назначать обработчик события одного объекта другому объекту или его событию, если совпадают типы событий. Подобная возможность обеспечивается с помощью *указателя на класс*. Кроме явно задаваемых параметров, например, параметра Sender, методу всегда передается указатель на вызвавший его экземпляр класса. Этим указателем является параметр Self.

Для посылки сообщения оконным элементам управления можно использовать функцию SendMessage. Посылка сообщения может понадобиться в случае, когда компонент через свои свойства не предоставляет всех своих возможностей. Например, список ListBox не имеет свойств, напрямую управляющих горизонтальной полосой прокрутки. Поэтому для отображения и скрытия горизонтальной полосы прокрутки можно послать списку соответствующее сообщение.

Функция SendMessage(hwnd: HWND; Msg: Cardinal; WParam, LParam: Longint): Longint посылает сообщение оконному элементу управления, ссылка (дескриптор) на который задана параметром hwnd. В Delphi дескриптор оконного элемента содержит свойство Handle. Параметр Msg указывает код сообщения, а параметры WParam и LParam содержат дополнительную информацию о сообщении, и их значения зависят от конкретного сообщения.

Рассмотрим следующий пример:

```
Label1.Caption:=IntToStr(SendMessage(ListBox1.Handle, LB_GetCount, 0, 0));
```

Здесь списку ListBox1 посылается сообщение LB_GetCount, которое предписывает списку выдать число его элементов. Для доступа к списку используется его дескриптор, значение которого содержит свойство Handle. Так как для данного сообщения дополнительная информация не требуется, значения двух последних параметров равны нулю.

Отметим, что число элементов списка можно получить также через подсвойство Count свойства Items списка.

Динамическая информация о типе

Объекты содержат динамическую информацию о собственном типе (RTTI, Run-Time Type Information) и наследовании, которая доступна во время выполнения программы и

которую можно использовать, например, для проверки принадлежности объекта к тому или иному типу. Поскольку для каждого объекта допустимы только определенные операции, зависящие от его типа, такая проверка позволяет предотвратить опасные ситуации, связанные с выполнением недопустимых действий.

Большинству методов при вызове передается параметр `Sender`, имеющий тип `TObject`. Для выполнения с этим параметром операций, таких как, например, вызов метода или присваивание значения свойству, его необходимо привести к типу того объекта, для которого выполняются эти операции. Различают *явное* и *неявное* приведение (преобразование) типов.

Для операций с типами в языке Object Pascal служат инструкции `is` и `as`. Инструкция `is` используется в выражении

```
<Объект> is <Класс>
```

и проверяет, принадлежит ли объект указанному классу или одному из его потомков. Если да, то это выражение имеет значение `True`, что указывает на совместимость типов. В противном случае выражение имеет значение `False`.

Инструкция `as` предназначена для приведения одного типа к другому и используется в выражении вида

```
<Объект> as <Класс>
```

В этом выражении объект приводится к типу класса, такое приведение типа является *неявным*.

Рассмотрим следующий пример неявного приведения типа:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if (Sender is TButton) then (Sender as TButton).Caption := TimeToStr(Now);
end;
```

Здесь при нажатии кнопки `Button1` в ее заголовке отображается текущее время. Для доступа к объекту кнопки и его свойству `Caption` используется параметр `Sender`, тип которого приводится к типу `TButton` кнопки. Предварительно выполняется проверка, можно ли выполнить подобное приведение типа.

Если обработчик предназначен только для кнопки `Button1`, то изменение заголовка кнопки проще выполнить с помощью инструкции вида

```
Button1.Caption:=TimeToStr(Now);
```

Использование параметра `Sender` и, соответственно, приведение типа может оказаться необходимым в случаях, когда процедура обработки является общей для нескольких компонентов, в том числе разных типов.

Явное приведение типа выполняется с помощью следующей конструкции:

```
<Тип>(<Объект>)
```

Пример явного приведения типа:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  TButton(Sender).Caption:='Кнопка';
end;
```

Заголовок компонента, нажатого пользователем, заменяется заголовком `Кнопка`. Для выполнения присваивания тип компонента приводится к типу `TButton`.

Библиотека визуальных компонентов

Библиотека визуальных компонентов (Visual Component Library, VCL) содержит большое количество классов, предназначенных для быстрой разработки приложений. Библиотека написана на Object Pascal и непосредственно связана с интегрированной средой разработки приложений Delphi. Несмотря на название, в VCL содержатся главным образом невизуальные компоненты, однако имеются и визуальные, а также другие классы, начиная с абстрактного класса `TObject`. При этом все компоненты являются классами, но не все классы являются компонентами.

Как отмечалось в *главе 1*, в Delphi 7 также имеется возможность использовать библиотеку CLX (межплатформенный вариант библиотеки VCL) для разработки приложений под Windows и Linux.

Все классы VCL расположены на определенном уровне иерархии и образуют дерево (иерархию) классов. Знание происхождения объекта оказывает значительную помощь при его изучении, т. к. потомок наследует все элементы объекта-родителя. Так, если свойство `Caption` принадлежит классу `TControl`, то это свойство будет и у его потомков, например, у классов `TButton` и `TCheckBox`, и у компонентов — кнопки `Button` и флажка `CheckBox` соответственно. Фрагмент иерархии классов с важнейшими классами показан на рис. 2.2. Полная схема иерархии рассматриваемых в книге классов приведена в *приложении 1*.

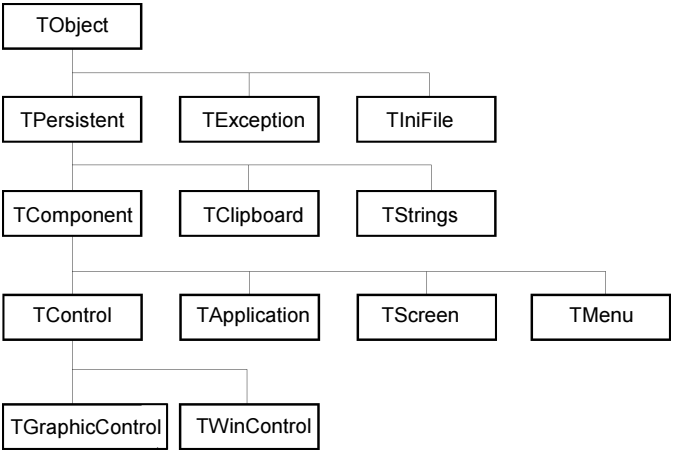


Рис. 2.2. Фрагмент иерархии классов

Кроме иерархии классов, большим подспорьем в изучении системы программирования являются исходные тексты модулей, которые находятся в каталоге SOURCE главного каталога Delphi.

Класс `TObject` — общий предок всех классов Object Pascal — находится в корне иерархии. Этот класс является абстрактным и реализует наиболее *общие* для всех классов-потомков методы, важнейшими из которых являются:

- `Create` (создание объекта);
- `Destroy` (удаление объекта);
- `Free` (удаление объекта, созданного методом `Create`, при этом вызывается и метод `Destroy`).

Большинство этих методов переопределяются в классах-потомках. Дадим краткую характеристику важнейшим классам-потомкам `TPersistent`, `TComponent` и `TControl`, к которым относится большинство общих свойств, методов и событий и которые, в свою очередь, также порождают множество классов.

Класс `TPersistent` является абстрактным классом для тех объектов, свойства которых загружаются из потока и сохраняются в потоке. Механизм потоков используется для работы с памятью (обычно дисковой или оперативной). В дополнение к методам класса `TObject` класс `TPersistent` имеет метод `Assign`, позволяющий передавать поля и свойства одного объекта другому.

Класс `TComponent` является базовым для всех компонентов и в дополнение к методам своих предков предоставляет средства, благодаря которым компоненты способны владеть другими компонентами. В результате при помещении в форму любой компонент будет принадлежать другому компоненту (чаще всего форме). При создании компонента он обеспечивает автоматическое создание всех принадлежащих ему компонентов, а при удалении этого компонента все принадлежащие ему компоненты также автоматически удаляются. Отметим некоторые свойства и методы класса `TComponent`.

Свойства:

- `Components` (список принадлежащих компонентов);
- `ComponentCount` (число принадлежащих компонентов);
- `ComponentIndex` (номер компонента в списке принадлежащих компонентов);
- `ComponentState` (состояние текущего компонента);
- `Name` (имя компонента);
- `Owner` (владелец компонента);
- `Tag` (целое значение, хранимое вместе с компонентом).

Методы:

- `DestroyComponents` (разрушает все принадлежащие компоненты);
- `Destroying` (уведомляет принадлежащий компонент о его разрушении);
- `FindComponent` (находит компонент в списке `Components`).

От класса `TComponent` происходят *визуальные* и *невизуальные* компоненты. Многие невизуальные компоненты порождены непосредственно от класса `TComponent`, например, таймер (`Timer`).

Компонент `Timer` позволяет реализовать выполнение действий через определенные интервалы времени. Для этого используется событие `OnTimer` типа `TNotifyEvent`, генерируемое по истечении заданного интервала времени. Интервал времени в миллисекундах задается свойством `Interval` типа `Cardinal` и по умолчанию равен 1000, что соот-

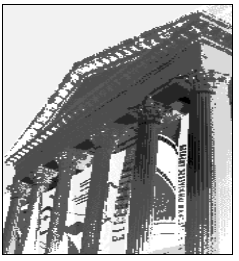
ветствует тиканью таймера через 1 секунду. Если установить интервал равным нулю, то таймер остановится и не будет генерировать события `OnTimer`. Остановить таймер также можно, присвоив значение `False` свойству `Enabled`.

Замечание

Если система занята выполнением более важных задач, чем отслеживание хода таймера, то интервал между соседними тиканьями таймера может увеличиваться. Это происходит, например, при считывании с диска больших файлов или при выполнении длинного цикла.

Класс `TControl` является базовым классом для *визуальных* компонентов (элементов управления) и обеспечивает основные средства для их функционирования, в том числе прорисовку на экране. Все визуальные компоненты делятся на *оконные* и *графические*, происходящие, соответственно, от классов `TWinControl` и `TGraphicControl`.

Таковы наиболее общие классы библиотеки визуальных компонентов. В следующих главах книги мы рассмотрим большое количество других классов и компонентов, а также наиболее важные характеристики классов `TObject`, `TPersistent`, `TComponent` и `TControl`.



ГЛАВА 3

Использование визуальных компонентов

Для создания интерфейса приложений система Delphi предлагает обширный набор визуальных компонентов, основные из которых располагаются на страницах **Standard**, **Additional** и **Win32** Палитры компонентов. Их называют *стандартными*, *дополнительными* и *32-разрядными* (введенными в Windows 95) компонентами соответственно.

Такое деление компонентов исходит скорее из названия страниц, чем из их функционального назначения или важности, поскольку грань, например, между стандартными и дополнительными элементами управления довольно нечеткая. Так, кнопки `Button` и `BitBtn`, располагаясь на разных страницах, практически не отличаются по функциям.

На странице **Standard** (рис. 3.1) Палитры компонентов находятся интерфейсные компоненты, большинство из которых использовалось еще в первых версиях Windows:

- `Frames` (фреймы);
- `MainMenu` (главное меню);
- `PopupMenu` (всплывающее меню);
- `Label` (надпись);
- `Edit` (однострочный редактор);
- `Memo` (многострочный редактор);
- `Button` (стандартная кнопка);
- `CheckBox` (независимый переключатель, флажок);
- `RadioButton` (зависимый переключатель, переключатель);
- `ListBox` (список);
- `ComboBox` (поле со списком);
- `ScrollBar` (полоса прокрутки);
- `GroupBox` (группа);
- `RadioGroup` (группа зависимых переключателей);

- `Panel` (панель);
- `ActionList` (список действий).

Рис. 3.1. Страница **Standard** Палитры компонентов

Компоненты перечислены последовательно в порядке расположения их значков на странице. Первый значок соответствует не компоненту, а инструменту (стрелке) отмены выбора компонента на странице.

На странице **Additional** (рис. 3.2) Палитры компонентов находятся следующие компоненты:

- `BitBtn` (кнопка с рисунком);
- `SpeedButton` (кнопка быстрого доступа);
- `MaskEdit` (однострочный редактор с вводом по шаблону);
- `StringGrid` (таблица строк);
- `DrawGrid` (таблица);
- `Image` (графическое изображение);
- `Shape` (геометрическая фигура);
- `Bevel` (фаска);
- `ScrollBox` (область прокрутки);
- `CheckListBox` (список флажков);
- `Splitter` (разделитель);
- `StaticText` (статический текст);
- `ControlBar` (контейнер для панели инструментов);
- `ApplicationEvents` (события приложения);
- `ValueListEditor` (редактор списка значений);
- `LabeledEdit` (однострочный редактор с надписью);
- `ColorBox` (комбинированный список выбора цвета);
- `Chart` (диаграмма);
- `ActionManager` (менеджер действий);
- `ActionMainMenuBar` (главное меню действий);
- `ActionToolBar` (панель действий);

Рис. 3.2. Страница **Additional** Палитры компонентов

- `XPColorMap` (цветовая карта XP для меню и панелей инструментов);
- `StandardColorMap` (стандартная цветовая карта для меню и панелей инструментов);
- `TwilightColorMap` (цветовая карта Twilight для меню и панелей инструментов);
- `CustomizeDlg` (диалог настройки).

Компоненты `XPColorMap`, `StandardColorMap` и `TwilightColorMap` появились впервые в Delphi 7.

На странице **Win32** (рис. 3.3) Палитры компонентов находятся компоненты, относящиеся к 32-разрядному интерфейсу Windows:

- `TabControl` (вкладка);
- `PageControl` (блокнот);
- `ImageList` (список графических изображений);
- `RichEdit` (полнофункциональный тестовый редактор);
- `TrackBar` (ползунок);
- `ProgressBar` (индикатор выполнения);
- `UpDown` (счетчик);
- `HotKey` (редактор комбинаций "горячих" клавиш);
- `Animate` (просмотр видеоклипов);
- `DateTimePicker` (строка ввода даты);
- `MonthCalendar` (календарь);
- `TreeView` (дерево объектов);
- `ListView` (список);
- `HeaderControl` (разделитель);
- `StatusBar` (строка состояния);
- `ToolBar` (панель инструментов);
- `CoolBar` ("крутая" панель инструментов);
- `PageScroller` (прокрутка изображений);
- `ComboBoxEx` (расширенный комбинированный список);
- `XPManifest` (декларация XP).

Последний из компонентов страницы **Win32** впервые появился в Delphi 7.

Рис. 3.3. Страница компонентов **Win32**

Рассмотрим визуальные компоненты, наиболее часто используемые в качестве элементов управления приложений.

Общая характеристика визуальных компонентов

В библиотеке визуальных компонентов VCL для всех компонентов, в том числе и для предназначенных для работы с данными, базовым является класс `TControl`. Он обеспечивает основные функциональные атрибуты, такие как положение и размеры элемента, его заголовок, цвет и другие параметры. Класс `TControl` включает в себя общие для визуальных компонентов свойства, события и методы. В целом визуальные компоненты можно разделить на две большие группы: оконные и неоконные элементы управления.

Оконный элемент управления представляет собой специализированное окно, предназначенное для решения конкретной задачи. К таким элементам относятся, например, кнопки, текстовые поля, полосы прокрутки. Для оконных элементов управления базовым классом является `TWinControl` — прямой потомок класса `TControl`.

На получение фокуса ввода оконные элементы управления могут реагировать двумя способами:

- с помощью курсора редактирования;
- с помощью прямоугольника фокуса.

Такие компоненты, как редакторы `Edit`, `DBEdit`, `Memo` или `DBMemo`, при получении фокуса ввода отображают в своей области курсор редактирования (текстовый курсор). По умолчанию курсор редактирования имеет вид мигающей вертикальной линии и показывает текущую позицию вставки вводимых с клавиатуры символов. Курсор редактирования перемещается с помощью клавиш управления курсором.

Компоненты, не связанные с редактированием текста, получение фокуса ввода обычно отображают с помощью пунктирного черного прямоугольника. При этом, например, для кнопки `Button` этот прямоугольник появляется вокруг ее заголовка, а для списков `ListBox` и `DBListBox` прямоугольник выделяет выбранную в текущий момент времени строку. Выбранная строка может окрашиваться в какой-либо цвет, чаще всего синий.

Оконные элементы управления содержат дескриптор (определитель) окна (window handle). *Дескриптором окна* в операционной системе Windows называется 32-битная величина, однозначно определяющая данное окно. Приложение использует этот определитель для обращения к окну.

Кроме того, оконные элементы управления могут содержать другие элементы управления, выступая в роли класса-контейнера, являющегося родительским по отношению к находящимся в нем объектам.

Для неоконных элементов управления базовым является класс `TGraphicControl`, производимый непосредственно от класса `TControl`. Неоконные элементы управления не могут получать фокус ввода и быть родителями других элементов пользовательского интерфейса. Достоинством неоконных элементов управления по сравнению с оконными является меньшее расходование ресурсов, т. к. для них не нужен дескриптор окна. Неоконными элементами управления являются, например, компоненты `Label` и `DBText`.

Рассмотрим подробнее общие свойства, события и методы визуальных компонентов, а также класс `TStrings`, который является базовым классом для операций со строковыми данными.

Свойства

Свойства позволяют управлять внешним видом и поведением компонентов при проектировании и выполнении приложения. Свойства компонентов, доступные при проектировании приложения, также доступны при его выполнении. Вместе с тем есть свойства, которые доступны только во время выполнения приложения. Обычно большинство значений свойств компонентов устанавливается на этапе проектирования с помощью Инспектора объектов, однако для наглядности в приводимых нами примерах свойствам часто присваиваются значения с помощью инструкции присваивания.

Свойства можно разделить на следующие группы (в скобках приведены названия групп в окне Инспектора объектов):

- действие (**Action**);
- операции с базами данных (**Database**);
- перемещение и стыковка компонентов (**Drag**, **Drop and Docking**);
- входные (**Input**);
- контекстная помощь (**Help and Hints**);
- макет (**Layout**);
- наследование (**Legacy**);
- связи (**Linkage**);
- местные (**Locale**);
- локализация (**Localizable**);
- разное (**Miscellaneous**);
- визуальные (**Visual**).

В окне Инспектора объектов свойство может отображаться сразу в нескольких группах. Например, свойство `Visible` одновременно принадлежит группам **Action** и **Visual**, а `Caption` — группам **Action**, **Localizable** и **Visual**. При изменении значения свойства в одной группе также изменятся значения, отображаемые в других группах.

Рассмотрим наиболее общие свойства визуальных компонентов, описав свойства (кроме `Name`) в алфавитном порядке. Отметим, что отдельные компоненты имеют не все рассматриваемые далее свойства, например, редактор `Edit` не имеет свойства `Caption`, а надпись `Label` — свойства `ReadOnly`.

Свойство `Name` типа `TComponentName` указывает имя компонента, которое программист использует для управления компонентом во время выполнения приложения. Отметим, что тип `TComponentName` эквивалентен типу `String`. Каждый новый компонент, помещаемый в форму, получает имя по умолчанию, автоматически образуемое путем добавления к названию компонента его номера в порядке помещения в форму. Например, первый однострочный редактор `Edit` получает имя `Edit1`, второй — `Edit2` и т. д.

На этапе разработки приложения программист может изменить имя компонента по умолчанию на более осмысленное и соответствующее назначению компонента. Существует несколько точек зрения по поводу присвоения имен компонентам. Согласно одной из них, имя рекомендуется составлять из назначения компонента и его названия. Другим вариантом будет указание в имени вместо названия компонента его префикса. Префикс является сокращением названия, например, для однострочного редактора `Edit` префикс может быть `edt`, для надписи `Label` — `lbl`, для формы `Form` — `fm`. То есть однострочный редактор, предназначенный для ввода фамилии сотрудника, можно назвать `NameEdit` или `edtName`. Оба способа одинаково допустимы, и на практике каждый разработчик использует тот, который для него наиболее удобен, или вообще именует компоненты так, как ему хочется.

Для наглядности мы будем использовать в наших примерах в качестве имен визуальных и невизуальных компонентов их имена по умолчанию, например, `Label1`, `Edit2` или `Button3`.

При *динамическом* создании компонентов во время выполнения приложения они тоже автоматически получают имена по умолчанию. Разработчик может изменить имя нового компонента, программно установив нужное значение его свойства `Name`.

Приведем пример создания компонента во время выполнения программы:

```
with Edit.Create(Self) do begin
      Parent := Form1;
      Name := 'edtName';
      Text := 'Иванов П.О.';
      Left := 100;
      Top := 60;
end;
```

Здесь динамически создается однострочный редактор `Edit`, который получает имя `edtName`. Новому редактору устанавливается текстовое значение и координаты его размещения в контейнере — владельце этого компонента. Владельцем нового редактора является форма `Form1`.

Организовать совместное использование нескольких взаимосвязанных элементов управления можно с помощью специального компонента `ActionList`. Он предназначен для централизованного управления различными элементами, например такими, как кнопка `Button` и пункт меню `MenuItem`. Связь между элементом управления и объектом действия, содержащимся в компоненте `ActionList`, осуществляется через свойство `Action` типа `TBasicAction` элемента управления.

Рассмотрим на примере, как устанавливается связь с объектом действия:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.Action := Action1;
end;
```

Здесь кнопка `Button1` связана с объектом действия `Action1`. При нажатии кнопки `Button1` будет вызван не обработчик события `OnClick` кнопки, а процедура обработки события `OnExecute` объекта `Action1`.

Свойство `Align` типа `TAlign` определяет *способ выравнивания* компонента внутри контейнера, в котором он находится. Чаще всего в роли такого контейнера выступает форма (`Form`) или панель (`Panel`). Выравнивание используется в случаях, когда требуется, чтобы какой-либо интерфейсный элемент занимал определенное положение относительно содержащего его контейнера независимо от изменения размеров последнего.

Свойство `Align` может принимать следующие значения:

- `alNone` — выравнивание не используется, компонент по умолчанию находится на том месте, куда был помещен при разработке приложения;
- `alTop` — компонент перемещается в верхнюю часть контейнера, высота компонента не меняется, а его ширина становится равной ширине контейнера;
- `alBottom` — аналогично действию `alTop`, но компонент перемещается в нижнюю часть контейнера;
- `alLeft` — компонент перемещается в левую часть контейнера, ширина компонента не меняется, его высота становится равной высоте контейнера;
- `alRight` — аналогично действию `alLeft`, но компонент перемещается в правую часть контейнера;
- `alClient` — компонент занимает всю поверхность контейнера;
- `alCustom` — размеры и положение компонента в контейнере устанавливаются разработчиком.

Например, выравнивание панели относительно формы выполняется с помощью следующего кода:

```
Panel1.Align := alClient;
```

В форму помещается панель `Panel1`, которая должна занять всю клиентскую область формы. С этой целью для ее свойства `Align` устанавливается значение `alClient`.

Свойство `Caption` типа `TCaption` содержит строку для *заголовка* компонента. Отметим, что тип `TCaption` равен типу `String`. Отдельные символы в заголовке могут быть подчеркнуты, они обозначают комбинации клавиш быстрого доступа: нажатие клавиши с указанным символом при нажатой клавише <Alt> вызывает то же действие, что и щелчок мышью на элементе управления с этим заголовком. Для определения комбинации клавиш необходимо поставить в заголовке перед соответствующим символом знак `&`, например:

```
CheckBox1.Caption := 'во&Зврат тары';          <Alt>+<З>
RadioGroup1.Caption := '&Conditions';          <Alt>+<C>
```

Замечание

При реагировании на комбинации клавиш Windows учитывает раскладку клавиатуры, поэтому пользователь должен не забывать переключать язык, например, с русского на английский и наоборот.

Свойство `Color` типа `TColor` определяет *цвет фона* (поверхности) компонента. Тип `TColor` описан следующим образом:

```
type TColor = -(COLOR_ENDCOLORS + 1) .. $02FFFFFF;
```

Значение свойства Color представляет собой четырехбайтовое шестнадцатеричное число. Старший байт указывает палитру и обычно имеет значение $00, что соответствует отображению цвета, наиболее близкого к задаваемому свойством Color. Младшие три байта задают RGB-интенсивности (интенсивности базовых красного, зеленого и синего цветов), которые при смешивании дают требуемый цвет. Когда значение байта, содержащего код интенсивности, равно $FF, соответствующий базовый цвет имеет максимальную интенсивность, если значение байта равно $00, то соответствующий базовый цвет выключен. Отсутствие базовых цветов приводит к черному цвету, а их максимальная интенсивность образует белый цвет.

Таким образом, черному цвету соответствует код $000000, белому — $FFFFFF, красному — $0000FF, зеленому — $00FF00, синему — $FF0000.

Часто удобно задавать цвета с помощью констант. Отображаемый цвет зависит от параметров видеокарты и монитора, в первую очередь от установленного цветового разрешения. При использовании констант, приведенных в табл. 3.1, отображается цвет, наиболее близкий к указанному константой. Все константы, кроме clDkGray и clLtGray, можно выбирать с помощью Инспектора объектов. Дополнительно *во время выполнения* приложения можно использовать константы clDkGray и clLtGray, которые дублируют значения clGray и clSilver соответственно.

Таблица 3.1. *Константы основных цветов*

Константа	Цвет	Значение
clAqua	Ярко-голубой	$FFFF00
clBlack	Черный	$000000
clBlue	Синий (голубой)	$FF0000
clCream	Кремовый	$F0FBFF
clFuchsia	Пурпурный (фуксия)	$FF00FF
clGray	Серый	$808080
clGreen	Зеленый	$008000
clLime	Ярко-зеленый	$00FF00
clMaroon	Коричневый (каштановый)	$000080
clMedGray	Средне-серый	$A4A0A0
clMoneyGreen	Денежно-зеленый	$C0DCC0
clNavy	Темно-синий	$800000
clOlive	Оливковый	$008080
clPurple	Фиолетовый	$800080
clRed	Красный	$0000FF
clSilver	Серебряный	$C0C0C0
clSkyBlue	Небесно-голубой	$F0CAA6
clTeal	Сине-зеленый	$808000

Таблица 3.1 *(окончание)*

Константа	Цвет	Значение
clWhite	Белый	$FFFFFF
clYellow	Желтый	$00FFFF

Другой набор констант (табл. 3.2) указывает цвета в составе системной палитры Windows, установленные на вкладке **Appearance** (Оформление) диалогового окна **Display Properties** (Свойства экрана). Определяемый такой константой цвет зависит от выбранной цветовой схемы.

Таблица 3.2. *Константы системных цветов Windows*

Константа	Элемент, для которого определяется цвет
clBackground	Фон окна
clActiveCaption	Заголовок активного окна
clInactiveCaption	Заголовок неактивного окна
clMenu	Фон меню
clWindow	Фон Windows
clWindowFrame	Рамки окна
clMenuText	Текст меню
clWindowText	Текст внутри окна
clCaptionText	Текст заголовка активного окна
clInactiveCaptionText	Текст заголовка неактивного окна
clActiveBorder	Рамка активного окна
clInactiveBorder	Рамка неактивного окна
clAppWorkSpace	Рабочая область приложения
clHighlight	Фон выделенного текста
clHightlightText	Выделенный текст
clBtnFace	Кнопка
clBtnShadow	Тень кнопки
clGrayText	Неактивный интерфейсный элемент
clBtnText	Текст кнопки
clBtnHighlight	Подсвеченная кнопка
clScrollBar	Полоса прокрутки
cl3DDkShadow	Теневая сторона объемных элементов
cl3DLight	Яркая сторона объемных элементов
clInfoText	Текст информационных средств
clInfoBk	Фон информационных средств

Таблица 3.2 *(окончание)*

Константа	Элемент, для которого определяется цвет
clGradientActiveCaption	Правая сторона градиентного цвета заголовка активного окна, левую сторону цвета определяет clActiveCaption
clGradientInactiveCaption	Правая сторона градиентного цвета заголовка неактивного окна, левую сторону определяет clInactiveCaption
clDefault	Цвет по умолчанию

Константы cl3DDkShadow, cl3DLight, clInfoText и clInfoBk в системах будут доступны только Windows 95/NT, а константы clGradientActiveCaption и clGradientInactiveCaption — только для Windows 98/2000.

В ряде случаев требуется ограничить *размер* элемента управления, например, чтобы запретить пользователю устанавливать его меньше определенного предела. Это можно выполнить программно, проверяя размеры компонента при их изменении и при необходимости корректируя соответствующим образом. Кроме того, для задания *ограничений на размер* компонентов имеется свойство Constraints типа TSizeConstraints, присутствующее у многих визуальных компонентов, в том числе у формы.

Тип TSizeConstraints является классом с собственными свойствами. Наиболее важные из них — MinHeight, MaxHeight, MinWidth и MaxWidth, задающие, соответственно, минимальные и максимальные возможные высоту и ширину управляющего элемента. Эти свойства класса TSizeConstraints являются подсвойствами для Constraints и имеют тип TConstraintSize (TConstraintSize = 0 .. MaxInt).

Свойство Ctl3D типа Boolean задает *вид* визуального компонента. Если свойство Ctl3D имеет значение False, то компонент отображается плоским, если True — то трехмерным (по умолчанию). Эффект трехмерности (объемности) создается углублением объекта относительно поверхности контейнера. Это свойство относится не ко всем компонентам, например, компонент Label1 его не имеет.

Свойство Cursor типа TCursor определяет *вид указателя мыши* при размещении его в области компонента. Delphi предлагает более двадцати предопределенных видов указателя мыши и соответствующих им констант, основными из которых являются следующие:

- crDefault (вид по умолчанию, обычно стрелка);
- crNone (указатель не виден);
- crArrow (стрелка);
- crCross (крест);
- crDrag (стрелка с листом бумаги);
- crHourGlass (песочные часы).

Кроме того, можно самостоятельно разрабатывать новые рисунки для указателя мыши, сохраняя их в соответствующих файлах ресурсов (cur, res). Создать изображение указателя мыши можно в любом графическом редакторе, в том числе в Image Editor версии 3.0, поставляемом с Delphi. При выполнении программы изображение указателя

загружается из файла, а свойству `Cursor` управляемого компонента присваивается значение номера изображения.

Свойство `DragCursor` типа `TCursor` определяет *вид указателя мыши* при перемещении компонентов. Значения этого свойства не отличаются от значений свойства `Cursor`. По умолчанию свойству устанавливается значение `crDrag`.

Свойство `DragMode` типа `TDragMode` используется при программировании операций, связанных с *перемещением объектов* способом drag-and-drop ("перетаскивание"), и определяет поведение элемента управления при его перемещении мышью. Свойство `DragMode` может принимать одно из двух значений: `dmAutomatic` и `dmManual`. По умолчанию оно имеет значение `dmManual`, и элемент управления перемещать нельзя, пока не будет вызван метод `BeginDrag`. Если этому свойству задать значение `dmManual`, то элемент управления можно перемещать мышью в любой момент. Кроме установки для свойства `DragMode` требуемого значения, программист должен написать код действий, управляющих перемещением элемента, т. е. подготовить обработчики событий, связанных с операцией перемещения.

Свойство `Enabled` типа `Boolean` определяет *активность* компонента, т. е. его способность реагировать на поступающие сообщения, например, от мыши или клавиатуры. Если свойство имеет значение `True` (по умолчанию), то компонент активен, в противном случае нет. Неактивное состояние выделяется цветом, при этом заголовок или текст неактивного компонента становятся бледными.

Компонент может быть отключен (заблокирован), например, в случае, когда пользователю запрещено изменять значение поля записи с помощью редактора `Edit`. Блокировку компонента можно выполнить следующим образом:

```
Edit1.Enabled := False;
```

Замечание

Блокировка любого визуального компонента с использованием свойства `Enabled` относится только к пользователю. Программно можно изменить его значение, например, с помощью следующей инструкции присваивания:

```
Edit1.Text := 'Иванов П.А.';
```

Запретить изменение значения компонента можно также, установив значение `True` свойства `ReadOnly`.

Свойство `Font` типа `TFont` определяет *шрифт* текста, отображаемого на визуальном компоненте. В свою очередь, класс `TFont` содержит свойства, позволяющие управлять параметрами шрифта. Ниже перечисляются основные свойства класса `TFont`.

- Свойство `Name` типа `TFontName` определяет название шрифта, например, Arial или Times New Roman. Отметим, что свойство `Name` шрифта не связано с одноименным свойством самого компонента.
- Свойство `Size` типа `Integer` задает размер шрифта в пунктах (пункт равен 1/72 дюйма).
- Свойство `Height` типа `Integer` задает размер шрифта в пикселах. Если значение этого свойства является положительным числом, то в него включается и межстрочный интервал. Если размер шрифта имеет отрицательное значение, то интервал не учитывается.

- Свойство `Style` типа `TFontStyle` задает стиль шрифта и может принимать комбинации следующих значений:
  - `fsItalic` (курсив);
  - `fsBold` (полужирный);
  - `fsUnderline` (подчеркнутый);
  - `fsStrikeOut` (перечеркнутый).
- Свойство `Color` типа `TColor` управляет цветом текста.

Свойства `Size` и `Height` взаимозависимы, при установке значения одного из них значение второго изменяется автоматически.

Пример задания цвета компонента:

```
Edit1.Font.Color := clGreen;
Edit1.Color := clBlue;
```

Здесь для редактора `Edit1` устанавливаются зеленый цвет текста и синий цвет фона.

Как отмечалось ранее, оконные элементы управления имеют дескрипторы (определители). Доступ к дескриптору окна обеспечивает свойство `Handle` типа `HWND`. Обычно дескриптор окна используется при вызове API-функций Windows.

Свойства `Height` и `Width` типа `Integer` указывают соответственно вертикальный и горизонтальный *размеры* компонента в пикселах.

Свойства `Left` и `Top` типа `Integer` определяют *координаты* левого верхнего угла компонента относительно содержащего его контейнера, например, формы или панели. Отметим, что форма также является компонентом, для нее координаты отсчитываются от левого верхнего угла экрана монитора. Свойства `Left` и `Top` совместно с `Height` и `Width` задают положение и размер компонента.

Свойство `HelpContext` типа `THelpContext` задает *номер раздела* справочной системы. Если при выполнении программы компонент находится в фокусе ввода, то нажатие клавиши <F1> приводит к отображению на экране контекстной справки, связанной с данным компонентом.

Свойство `Hint` типа `String` задает *текст подсказки*, появляющийся, когда курсор находится в области компонента и некоторое время неподвижен. Подсказка представляет собой поле желтого (по умолчанию) цвета с текстом, поясняющим назначение или использование компонента. Для того чтобы подсказка отображалась, следует установить значение `True` свойства `ShowHint` типа `Boolean`. По умолчанию `ShowHint` имеет значение `False`, и подсказки не отображаются.

Свойство `PopupMenu` типа `TPopupMenu` указывает *локальное всплывающее меню*, появляющееся при размещении указателя в области компонента и одновременном нажатии правой кнопки мыши. Чтобы меню, ассоциированное с компонентом, появлялось при щелчке правой кнопкой мыши, нужно также задать значение `True` свойству `AutoPopup` типа `Boolean`. По умолчанию оно имеет значение `False`.

Свойство `Text` типа `TCaption` содержит строку, связанную с компонентом. Значение этого свойства является содержимым компонента. Например, для компонентов `Edit` и

`Memo` значение свойства `Text` отображается внутри них как редактируемые символьные данные.

Свойство `TabOrder` типа `TTabOrder` определяет *порядок получения* компонентами контейнера фокуса при нажатии клавиши <Tab>, т. е. последовательность номеров обхода (табуляции) компонентов. По умолчанию эта последовательность определяется при конструировании формы порядком помещения компонента в контейнер: для первого компонента свойство `TabOrder` имеет значение 0, для второго — 1 и т. д. Для изменения этого порядка нужно назначить соответствующие значения свойствам `TabOrder` компонентов контейнера. Порядок табуляции компонентов в контейнере не зависит от порядка табуляции компонентов в других контейнерах.

Два компонента не могут иметь одинаковые значения свойства `TabOrder`; система Delphi следит за этим, автоматически корректируя неправильные значения. Компонент, свойство `TabOrder` которого имеет значение 0, получает управление первым.

Свойство `TabOrder` используется совместно со свойством `TabStop` типа `Boolean`, указывающим на *возможность получения фокуса* компонентом. Если свойство `TabStop` имеет значение `True`, то элемент может получать фокус, если `False` — не может.

Изменять порядок табуляции визуальных компонентов можно также с помощью диалогового окна **Edit Tab Order** (Изменение порядка табуляции), при этом Delphi автоматически присваивает значения свойству `TabOrder` этих компонентов (рис. 3.4). Вызов диалогового окна осуществляется одноименной командой меню **Edit**.

Рис. 3.4. Окно управления порядком табуляции

Свойство `ReadOnly` типа `Boolean` определяет, разрешено ли связанному с вводом и редактированием текста элементу управления изменять находящийся в нем текст. Если свойство `ReadOnly` имеет значение `True`, то текст в элементе редактирования доступен только для чтения. Если свойство имеет значение `False` (по умолчанию), то текст можно редактировать. Запрет на редактирование относится только к пользователю, программным способом информация может быть изменена независимо от значения свойства `ReadOnly`, например, так:

```
Edit1.ReadOnly := True;
Edit1.Text := 'Новый текст';
```

Замечание

Даже если изменение содержимого редактора запрещено, элемент редактирования может получать фокус ввода. При получении фокуса ввода по-прежнему отображается мигающий курсор и разрешено перемещение по тексту, однако изменение содержимого редактора блокируется.

Объект поля также имеет свойство `ReadOnly`, разрешающее или запрещающее изменение его значения. Если свойство установлено в значение `True`, то программисту также запрещено изменять значение поля, и при попытке это сделать генерируется исключение.

Визуальные компоненты для таких свойств, как `Color`, `Ctl3D`, `Font` и `ShowHint`, могут принимать значения соответствующих свойств родительского элемента управления, например, формы. Источники значений для указанных свойств определяются следующими свойствами-признаками типа `Boolean`:

- `ParentColor` (цвет фона);
- `ParentCtl3D` (вид компонента);
- `ParentFont` (шрифт текста);
- `ParentShowHint` (признак отображения подсказки).

Большинство этих логических свойств по умолчанию имеют значение `True`, и компонент получает значения соответствующих параметров от родителя. Подобное наследование позволяет просто и удобно изменять значения параметров для многих компонентов одновременно: для этого достаточно установить соответствующее свойство родителя в нужное значение. Например, если для формы изменить размер шрифта на значение 12, то шрифт будет изменен и для всех компонентов, расположенных в ней и имеющих значение `True` свойства `ParentFont`.

Замечание

Если программист вручную изменяет для компонента какое-либо из наследуемых свойств, то соответствующий признак наследования автоматически сбрасывается в `False`. Таким образом, в дальнейшем компонент принимает для этого свойства собственное значение, а не родительское, и при необходимости наследования программист должен снова установить значение `True` для признака наследования.

Свойство `Parent` типа `TWinControl` указывает на *родительский элемент управления* для компонента. Родительский элемент является контейнером для размещения в нем других компонентов и отвечает за прорисовку всех подчиненных компонентов. Например, если расположенная в форме панель содержит редактор `Edit` и кнопку `Button`, то за своевременное и правильное отображение редактора и кнопки отвечает панель, а не форма, т. к. именно панель является владельцем указанных компонентов.

Пример использования свойства `Parent`:

```
procedure TForm1.Edit1DblClick(Sender: TObject);
begin
  Label1.Caption := (Edit1.Parent as TWinControl).Name;
end;
```

Здесь при двойном щелчке в поле редактирования `Edit1` в надписи `Label1` отображается имя его родительского компонента-контейнера. В частности, если редактор расположен на панели `Panel1`, то в надписи будет выведено `Panel1`.

При конструировании формы и размещении в ней различных компонентов для их свойства Parent автоматически устанавливается правильное значение, зависящее от того, в каком контейнере расположен компонент. Если же визуальный компонент создается во время выполнения программы, то программист должен самостоятельно установить значение его свойства Parent так, чтобы это значение указывало на контейнер, в котором этот компонент будет находиться и отображаться.

Рассмотрим следующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with TLabel.Create(Self) do begin
      Parent := Form1;
      Name := 'lblNew';
      Caption := 'New label';
      Left := 20;
      Top := 50;
  end;
end;
```

Здесь при нажатии кнопки Button1 динамически создается компонент типа TLabel, который размещается в форме Form1. Если не задать значение свойства Parent, то компонент будет создан, но на экране не появится. Кроме родителя, для компонента устанавливаются также значения свойств Name, Caption, Left и Top. Если не установить значение свойства Name, то компонент получит имя по умолчанию, например, Label1. Если не задать значение свойства Caption, то по умолчанию его значение совпадает со значением свойства Name.

У компонентов есть похожее на Parent свойство Owner типа TComponent, указывающее на *элемент-владелец* компонента. Обычно владельцем компонентов является форма, в которой они расположены. При удалении владельца удаляется и компонент.

Свойство Visible типа Boolean управляет видимостью компонента. Если для него установлено значение True, то компонент виден пользователю, при значении False компонент скрыт от пользователя. Отметим, что даже если компонент не виден, им можно управлять программно. Например:

```
Edit1.Visible := True;
Edit2.Visible := False;
```

Здесь однострочный редактор Edit1 устанавливается видимым пользователю, а однострочный редактор Edit2 скрывается.

События

Визуальные компоненты способны генерировать и обрабатывать достаточно большое число (несколько десятков) событий различных видов. К наиболее общим группам событий можно отнести следующие:

- выбор элемента управления;
- перемещение указателя мыши;
- вращение колеса мыши;

- нажатие клавиш;
- получение и потеря элементом управления фокуса ввода;
- перемещение объектов методом drag-and-drop (перетаскиванием).

Отметим, что в окне Инспектора объектов события сгруппированы по следующим группам:

- действие (**Action**);
- перемещение и стыковка компонентов (**Drag**, **Drop and Docking**);
- контекстная помощь (**Help and Hints**);
- входные (**Input**);
- среда (**Layout**);
- связь (**Linkage**);
- разное (**Miscellaneous**);
- визуальные (**Visual**).

Как говорилось, в окне Инспектора объектов событие может отображаться сразу в нескольких группах. Например, событие `OnResize` одновременно принадлежит группам **Layout** и **Visual**.

В языке Object Pascal — основе Delphi — события также являются свойствами и принадлежат к соответствующему типу. Большинство событий носят нотификационный (уведомляющий) характер и имеют тип `TNotifyEvent`:

```
type TNotifyEvent = procedure (Sender: TObject) of object;
```

Из этого описания видно, что нотификационные события содержат только источник события, на который указывает параметр `Sender`, и больше никакой информации не несут. Существуют и более сложные события, требующие передачи дополнительных параметров, например, событие, связанное с перемещением указателя мыши, передает координаты указателя.

При выборе элемента управления возникает событие `OnClick` типа `TNotifyEvent`, которое также называют *событием нажатия*. Обычно оно возникает при щелчке мышью на компоненте. При разработке приложений событие `OnClick` является одним из наиболее часто используемых.

Приведем в качестве примера процедуру обработки события выбора элемента `Edit1`:

```
procedure TForm1.Edit1Click(Sender: TObject);
begin
  Edit1.Color := Random($FFFFFF);
end;
```

Здесь при щелчке мышью в поле редактирования `Edit1` случайным образом изменяется цвет его фона.

Для некоторых компонентов событие `OnClick` может возникать и при других способах нажатия элемента управления, находящегося в фокусе ввода, например, для компонента `Button` — с помощью клавиши <Пробел> или <Enter>, а для компонента `CheckBox` — с помощью клавиши <Пробел>.

При щелчке любой кнопкой мыши генерируются еще два события: `OnMouseDown` типа `TMouseEvent`, возникающее при нажатии кнопки мыши, и `OnMouseUp` типа `TMouseEvent` — при отпускании кнопки.

При двойном щелчке левой кнопкой мыши в области компонента, кроме того, генерируется событие `OnDblClick` типа `TNotifyEvent`. События возникают в следующем порядке: `OnMouseDown`, `OnClick`, `OnMouseUp`, `OnDblClick`, `OnMouseDown`, `OnMouseUp`.

При *перемещении указателя мыши* над визуальным компонентом непрерывно вырабатывается событие `OnMouseMove` типа `TMouseMoveEvent`. Последний описан так:

```
type TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState;
                                 X, Y: Integer) of object;
```

В обработчике события параметр `Sender` указывает, над каким элементом управления находится указатель мыши, а параметры `X` и `Y` типа `Integer` определяют координаты (позицию) указателя. Координаты задаются относительно элемента управления, определяемого параметром `Sender`. Параметр `Shift` указывает на состояние клавиш <Alt>, <Ctrl> и <Shift> клавиатуры и кнопок мыши и может принимать комбинации следующих значений:

- `ssShift` — нажата клавиша <Shift>;
- `ssAlt` — нажата клавиша <Alt>;
- `ssCtrl` — нажата клавиша <Ctrl>;
- `ssLeft` — нажата левая кнопка мыши;
- `ssMiddle` — нажата средняя кнопка мыши;
- `ssDouble` — выполнен двойной щелчок мышью.

При нажатии любой из указанных клавиш к параметру `Shift` добавляется соответствующее значение. Например, если нажата комбинация клавиш <Shift>++<Ctrl>, то значением параметра `Shift` является `[ssShift, ssCtrl]`. Если не нажата ни одна клавиша, то параметр `Shift` принимает пустое значение `[]`.

Рассмотрим следующий пример:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
                               X, Y: Integer);
begin
  Form1.Caption := 'Координаты указателя мыши: ' + IntToStr(X) +
    ' и ' + IntToStr(Y);
end;
```

При перемещении указателя мыши в пределах формы его координаты отображаются в заголовке формы. Позиция указателя мыши отображается, если указатель находится в свободном месте формы, а не расположен над каким-либо элементом управления. Координаты `X` и `Y` отсчитываются в пикселах от левого верхнего угла формы, начиная с нуля.

Ряд событий связан с вращением колеса мыши: `OnMouseWheel`, `OnMouseWheelDown` и `OnMouseWheelUp`.

При *вращении колеса мыши* вперед и назад генерируются соответственно события OnMouseWheelDown и OnMouseWheelUp типа TMouseWheelUpDownEvent. Последний описан следующим образом:

```
type TMouseWheelUpDownEvent = procedure (Sender: TObject; Shift:
       TShiftState; MousePos: TPoint; var Handled: Boolean) of object;
```

Как и в типе TMouseMoveEvent, параметры Sender и Shift задают визуальный компонент, получающий события мыши, и состояние управляющих клавиш клавиатуры и кнопок мыши. Параметр MousePos содержит координаты указателя мыши, а логический параметр Handled определяет, будет ли визуальный компонент сам обрабатывать полученное событие (значение True) или передаст его родительскому элементу (значение False).

В процессе *вращения колеса мыши* при нахождении ее указателя над визуальным компонентом непрерывно вырабатывается событие OnMouseWheel типа TMouseWheelEvent. Это событие объединяет функциональность событий OnMouseWheelDown и OnMouseWheelUp, а его тип описан как

```
type TMouseWheelEvent = procedure(Sender: TObject; Shift: TShiftState;
  WheelDelta: Integer; MousePos: TPoint; var Handled: Boolean) of object;
```

Параметры Sender, Shift, MousePos и Handled не отличаются от параметров, используемых в типе TMouseWheelUpDownEvent. Параметр WheelDelta указывает, на сколько условных единиц сдвинулось колесо, при этом положительное значение параметра соответствует вращению вперед, а отрицательное — вращению назад.

При *работе с клавиатурой* генерируются события OnKeyPress и OnKeyDown, возникающие при нажатии клавиши, а также событие OnKeyUp, возникающее при отпускании клавиши. При нажатии клавиши возникновение событий происходит в следующем порядке: OnKeyDown, OnKeyPress, OnKeyUp.

При удерживании клавиши нажатой непрерывно генерируется событие OnKeyDown, событие OnKeyUp возникает однократно после отпускания клавиши.

Событие OnKeyPress типа TKeyPressEvent генерируется при каждом *нажатии алфавитно-цифровых* клавиш. Обычно оно обрабатывается, когда требуется реакция на нажатие одной клавиши. Тип TKeyPressEvent описан так:

```
type TKeyPressEvent = procedure (Sender: TObject; var Key: Char) of object;
```

Параметр Key содержит код ASCII нажатой клавиши, который может быть проанализирован и при необходимости изменен. Если параметру Key задать значение #0, то это соответствует отмене нажатия клавиши.

Замечание

Обработчик события OnKeyPress не реагирует на нажатие управляющих клавиш, однако несмотря на это параметр Key содержит код символа с учетом регистра, который определяется состоянием клавиш <Caps Lock> и <Shift>.

В качестве примера рассмотрим обработчик события OnKeyPress редактора:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = '!' then Key := #0;
end;
```

Здесь при изменении содержимого редактора `Edit1` пользователю запрещен ввод символа `!`.

Для обработки управляющих клавиш, не имеющих ASCII-кодов, можно программно использовать события `OnKeyDown` и `OnKeyUp` типа `TKeyEvent`, возникающие при нажатии любой клавиши.

Тип `TKeyEvent` описан так:

```
type TKeyEvent = procedure(Sender: TObject; var Key: Word;
                           Shift: TShiftState) of object;
```

Указанные события часто используются для анализа состояния управляющих клавиш <Shift>, <Ctrl>, <Alt> и др. Состояние этих клавиш и кнопок мыши указывает параметр `Shift`, который может принимать ранее рассмотренные значения. В отличие от события `OnKeyPress`, параметр `Key` имеет тип `Word`, а не `Char`, поэтому для преобразования находящегося в `Key` кода клавиши в символ можно использовать функцию `Chr()`.

Рассмотрим пример обработки нажатий управляющих клавиш:

```
Form1.KeyPreview := True;
...
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
    Shift: TShiftState);
begin
  if (Shift = [ssShift]) then begin
    if Key = VK_UP then begin
      UpDown1.Position := UpDown1.Position+1; Key := #0;
    end;
    if Key = VK_DOWN then begin
      UpDown1.Position := UpDown1.Position-1; Key := #0;
    end;
    if Key = VK_PRIOR then begin
      UpDown1.Position := UpDown1.Position+10; Key := #0;
    end;
    if Key = VK_NEXT then begin
      UpDown1.Position := UpDown1.Position-10; Key := #0;
    end;
  end;
end;
```

Здесь при совместном нажатии клавиши <Shift> и клавиш управления курсором соответственно изменяется значение обратного счетчика `UpDown1`. Указанные комбинации клавиш обрабатываются в обработчике события `onKeyDown` для формы `Form1`. Чтобы форма получала сообщения о нажатии клавиш в первую очередь, свойство формы `KeyPreview` установлено в значение `True`. Чтобы обработанное событие не передавалось дочернему компоненту формы, находящемуся в фокусе ввода, выполняется обнуление кода нажатой клавиши.

В обработчиках событий, связанных с нажатием клавиш, можно обрабатывать комбинации управляющих и алфавитно-цифровых клавиш, например, <Alt>+<S>. Рассмотрим еще один пример обработки нажатия клавиш, на этот раз управляющих и алфавитно-цифровых:

```
procedure TForm1.Edit2KeyDown(Sender: TObject; var Key: Word;
                              Shift: TShiftState);
begin
  if (Shift = [ssCtrl]) and (chr(Key) = '1') then
    MessageDlg('Нажаты клавиши <Ctrl> + <1> ', mtConfirmation, [mbOK], 0);
end;
```

Здесь при нахождении в фокусе ввода компонента `Edit2` нажатие комбинации клавиш <Ctrl>+<1> вызывает диалоговое окно **Confirm** с соответствующим сообщением.

Отдельные клавиши имеют особенности, например, при нажатии клавиши <Tab> не возникают события `OnKeyPress` и `OnKeyUp`.

При получении фокуса оконным элементом управления возникает событие `OnEnter` типа `TNotifyEvent`. Оно генерируется при активизации элемента управления любым способом, например, щелчком мыши или с помощью клавиши <Tab>. В случае потери фокуса ввода оконным элементом управления возникает событие `OnExit` типа `TNotifyEvent`.

Следующий пример демонстрирует, как производится обработка событий получения и потери фокуса элементом управления:

```
procedure TForm1.Edit1Enter(Sender: TObject);
begin
  Label1.Caption := (Sender as TControl).Name + ' активен';
end;

procedure TForm1.Edit1Exit(Sender: TObject);
begin
  Label1.Caption := TEdit(Sender).Name + ' не активен';
end;
```

В заголовке надписи `Label1` отображается активность (наличие или отсутствие фокуса) компонента `Edit1`. Доступ к свойству `Name` параметра `Sender` в процедурах обработки выполнен двумя способами. В первом случае параметр `Sender` с помощью конструкции `as` неявно приводится к типу `TControl`. Во втором случае параметр `Sender` явно приводится к типу `TEdit`.

Технология drag-and-drop ("перетаскивание") позволяет пользователю перемещать различные объекты, например, элементы одного списка в другой. При этом используются два элемента управления: источник и приемник. Источник содержит перемещаемый объект, а приемник — элемент управления, на который помещается элемент-источник.

С этой технологией перемещения объектов связаны события, перечисленные далее в порядке их возникновения.

- `OnStartDrag` типа `TStartDragEvent` генерируется источником в начале выполнения операции перемещения. Обработчику события передаются следующие параметры: объект-источник `Source` типа `Tobject` и объект-приемник `DragObject` типа `TdragObject`.
- `OnDragOver` типа `TDragOverEvent` вызывается приемником, когда перемещаемый объект находится над ним. Обработчику события передаются следующие параметры: объект-источник `Source` типа `TObject`, объект-приемник `Sender` типа `TObject`, теку-

щие координаты `X` и `Y` типа `Integer` указателя мыши, состояние перемещения `State` типа `TDragState` и признак `Accept` типа `Boolean` подтверждения операции перемещения. Состояние перемещения показывает, вошел ли перемещаемый объект в область приемника, передвигается ли он в этой области или покинул ее и не был отпущен. Анализ переданных параметров позволяет элементу-приемнику принять или отклонить элемент-источник. Если операция перемещения принята, то параметру `Accept` нужно установить значение `True`, в противном случае — `False`.

- `OnDragDrop` типа `TDragDropEvent` вызывается приемником, когда перемещаемый объект отпускается на нем. Для обработки операции перемещения обработчику события передаются исходный объект `Source` типа `TObject`, объект-приемник `Sender` и текущие координаты `X` и `Y` указателя мыши.
- `OnEndDrag` типа `TEndDragEvent` генерируется источником, когда операция перемещения завершается. Обработчику события передаются объект-приемник `Target` типа `TObject` и координаты `X` и `Y` точки, в которой был отпущен объект-источник `Sender`.

Замечание

Для событий `OnDragDrop` и `OnDragOver` параметр `Sender` представляет адресат операции (приемник), а для события `OnEndDrag` параметр `Sender` является перетаскиваемым объектом-источником.

Для реализации перетаскивания требуется написать обработчики указанных событий. Обычно достаточно обработать два события: `OnDragDrop` и `OnDragOver`. Для перетаскиваемого объекта-источника желательно установить свойство `DragMode` в значение `dmAutomatic`, при этом начало операции перемещения будет происходить автоматически, в противном случае нужно программно вызывать метод `BeginDrag`.

В обработчике события `OnDragOver` выполняется проверка, допустима ли операция перетаскивания. Если операцию перетаскивания можно принять, то признак `Accept` устанавливается в значение `True`, в противном случае — в значение `False`.

В обработчике события `OnDragDrop` осуществляются прием и обработка перемещенного объекта.

Рассмотрим пример, в котором компонент `Label1` перемещается в пределах формы `Form1`:

```
// Свойство DragMode компонента Label1 необходимо установить
// в значение dmAutomatic
procedure TForm1.FormDragOver(Sender, Source: TObject; X, Y: Integer;
                              State: TDragState; var Accept: Boolean);
begin
  if Source = Label1 then Accept := True else Accept := False;
end;

procedure TForm1.FormDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  Label1.Left := X;
  Label1.Top := Y;
end;
```

Перемещать оконные элементы управления во время выполнения приложения можно и без применения технологии drag-and-drop. Для этого используется событие OnMouseDown, связанное с нажатием кнопки мыши. Продемонстрируем, как это осуществляется на практике.

```
procedure TForm1.Label1MouseDown(Sender: TObject; Button: TMouseButton;
     Shift: TShiftState; X, Y: Integer);
begin
  // Перемещать компонент можно только при нажатии левой кнопки мыши
  if Button <> mbLeft then exit;
  // Передача компоненту потока сообщений мыши
  ReleaseCapture;
  // Перемещение компонента путем посылки ему соответствующего сообщения
  Edit1.Perform(WM_SysCommand, $F012, 0);
end;
```

Чтобы пользователь мог переместить компонент с помощью левой кнопки мыши, в начале процедуры-обработчика события выполняется проверка, нажата ли левая кнопка, в противном случае осуществляется выход из процедуры. При продолжении работы процедуры вызывается API-функция ReleaseCapture, которая обеспечивает получение компонентом потока сообщений, связанных с мышью. Собственно перемещение компонента DBEdit1 выполняет метод Perform, посылающий ему системное сообщение с кодом $F012.

В случае, когда указанным способом нужно перемещать более одного оконного элемента управления, удобно оформить для них общий обработчик события OnMouseDown, который может выглядеть следующим образом:

```
procedure TForm1.MoveWinControl(Sender: TObject; Button: TMouseButton;
     Shift: TShiftState; X, Y: Integer);
begin
  if Button <> mbLeft then exit;
  ReleaseCapture;
  (Sender as TWinControl).Perform(WM_SysCommand, $F012, 0);
end;
```

В предыдущих примерах выполнялось перемещение всего компонента. Более сложным является случай, когда перемещаются отдельные выбранные элементы компонента.

Рассмотрим пример перемещения элементов между списками, где источником является список ListBox1, а приемником — список ListBox2.

```
// Для списка ListBox1 свойству DragMode
// требуется задать значение dmAutomatic
procedure TForm1.ListBox2DragOver(Sender, Source: TObject; X, Y: Integer;
     State: TDragState; var Accept: Boolean);
begin
  if Source = ListBox1 then Accept := True else Accept := False;
end;

procedure TForm1.ListBox2DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
```

```
  with Source as TListBox do
      // Добавление элемента к списку ListBox2
      ListBox2.Items.Add(Items[ItemIndex]);
      // Удаление элемента из списка ListBox1
      Items.Delete(ItemIndex);
end;
```

Свойству `DragMode` списка-источника `ListBox1` задается значение `dmAutomatic`. В результате при нажатии левой кнопки мыши и размещении указателя на элементе списка операция перемещения начинается автоматически и не требует дополнительного программирования.

Для списка-приемника `ListBox2` подготовлены обработчики событий `DragOver` и `DragDrop`, возникающих в случае, когда перетаскиваемый элемент находится над списком-приемником и когда элемент отпускается на нем при отпускании левой кнопки мыши соответственно. В обработчике события `DragOver` проверяется, принадлежит ли перетаскиваемый элемент списку `ListBox1`, если да, то параметру `Accept` присваивается значение `True`. Это означает возможность принятия элемента списком `ListBox2`, что визуально отображается изменением вида указателя мыши. В противном случае элемент не может быть принят, и параметру `Accept` присваивается значение `False`.

В обработчике события `DragDrop` перетаскиваемый элемент добавляется к списку `ListBox2` и удаляется из списка `ListBox1`. Обращение к списку `ListBox1` выполнено с помощью конструкции `Source as TListBox`.

Использованная в примере инструкция `with` позволяет уменьшить объем текста и улучшить читаемость программы. Без ее использования инструкции обработчика события `OnDragDrop` выглядели бы так:

```
ListBox2.Items.Add((Source as TListBox).Items[(Source as
    TListBox).ItemIndex]);
(Source as TListBox).Items.Delete((Source as TListBox).ItemIndex);
```

В следующих примерах для лучшего восприятия текста программы инструкция `with` может отсутствовать, в этом случае имена пишутся полностью.

Если указатель некоторое время неподвижен в области компонента, то возникает событие `OnHint` типа `TNotifyEvent`, которое можно использовать для написания обработчиков, связанных с выводом контекстной помощи.

Методы

С визуальными компонентами, как и с другими объектами, связано большое количество методов, позволяющих создавать и удалять объекты, прорисовывать их, отображать и скрывать, а также выполнять другие операции. Рассмотрим методы, которые являются общими для всех визуальных компонентов.

Процедура `SetFocus` *устанавливает фокус ввода* на оконный элемент управления. Если элемент управления в данный момент времени не способен получить фокус ввода, то возникает ошибка. Поэтому при вероятном возникновении ошибки целесообразно предварительно выполнить соответствующую проверку. Проверить возможность активизации компонента позволяет функция `CanFocus: Boolean`, возвращающая значение

`True`, если элемент управления может получить фокус ввода, и `False` — в противном случае.

Элемент управления не может получать фокус ввода, если он находится в выключенном состоянии, и его свойство `Enabled` имеет значение `False`.

Так, перед получением компонентом `Edit1` фокуса ввода производится проверка возможности передачи ему фокуса:

```
If Edit1.CanFocus then Edit1.SetFocus;
```

Метод `Clear` служит для *очистки содержимого компонентов*, которые могут содержать текстовую информацию. Например:

```
ListBox1.Clear;
Memo1.Clear;
```

Метод `Refresh` используется для *обновления элемента управления*, состоящего в стирании имеющегося изображения элемента и его перерисовке. Обычно метод вызывается автоматически при необходимости перерисовки изображения. Принудительный вызов метода `Refresh` программным способом может понадобиться в случаях, когда программист сам управляет прорисовкой области визуального компонента, например, списка `ListBox`.

Метод `Refresh` автоматически вызывает методы `Invalidate` и `UpDate`. Метод `Invalidate` сообщает Windows, что изображение требует перерисовки. При первой возможности система выполняет эту операцию. Метод `UpDate` вызывает немедленную перерисовку указанного объекта. При необходимости эти методы можно вызывать непосредственно в программе.

Метод `Perform` предназначен для *посылки сообщений* оконным элементам управления. Использовать его удобнее, чем метод `SendMessage`, т. к. не нужно задавать параметр, содержащий ссылку на элемент управления. Функция `Perform(Msg: Cardinal; WParam, LParam: Longint): Longint` посылает сообщение, текст которого задается параметром `Msg`. Параметры `WParam` и `LParam` содержат дополнительную информацию о сообщении.

Например:

```
Label1.Caption := IntToStr(ListBox1.Perform(LB_GetCount, 0, 0));
```

Здесь списку `ListBox1` посылается сообщение `LB_GetCount`, заставляющее его вернуть число элементов. Результат выводится в надписи `Label1`.

Класс *TStrings*

Класс `TStrings` является базовым классом для операций со строковыми данными. Этот класс представляет собой контейнер для строк (коллекцию или массив строк). Для операций со строками класс `TStrings` предоставляет соответствующие свойства и методы. От класса `TStrings` происходит большое количество производных классов, например, `TStringList`, которые могут использоваться для задания различных типов строк.

Визуальные компоненты, способные работать со списком строк, имеют свойства, которые являются массивами строк, содержащихся в этих компонентах. Например, для списков `ListBox` и `DBListBox` и для групп зависимых переключателей `RadioGroup` и

`DBRadioGroup` таким свойством является `Items`, а для многострочных редакторов `Memo` и `DBMemo` — `Lines`. Указанные свойства для визуальных компонентов `ListBox` и `Memo` доступны при разработке и при выполнении приложения, а для визуальных компонентов `DBListBox` и `DBMemo`, связанных с данными, — только при выполнении приложения.

Рассмотрим особенности и использование класса `TStrings` на примере свойства `Items` списков. Работа с другими объектами типа `TStrings` происходит аналогично.

Каждый элемент списка является строкой, к которой можно получить доступ по ее номеру в массиве строк `Items`. Отсчет элементов списка начинается с нуля. Для обращения к первому элементу нужно указать `Items[0]`, ко второму — `Items[1]`, к третьему — `Items[2]` и т. д. При операциях с отдельными строками программист должен контролировать номера строк в списке и не допускать обращения к несуществующему элементу. Например, если список содержит три строки, то попытка работы с десятой строкой приведет к исключению.

Свойство `Count` типа `Integer` задает *число элементов* в списке. Поскольку первый элемент списка имеет нулевой номер, то номер последнего элемента равен `Count-1`.

Например, присваивание элементам списка `ListBox1` новых значений может быть реализовано так:

```
var n: integer;
...
for n := 0 to ListBox1.Items.Count – 1 do
  ListBox1.Items[n] := 'Строка номер ' + IntToStr(n);
```

Методы `Add` и `Insert` служат для *добавления/вставки строк* в список. Функция `Add(const S: string): Integer` добавляет заданную параметром `S` строку в конец списка, а в качестве результата возвращает положение нового элемента в списке. Процедура `Insert(Index: Integer; const S: String)` вставляет строку `S` в позицию с номером, определяемым параметром `Index`. При этом элементы списка, находившиеся до операции вставки в указанной позиции и ниже, смещаются вниз.

В приводимой далее процедуре к комбинированному списку `ComboBox1` добавляется строка `Нажата кнопка Button1`:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ComboBox1.Items.Add('Нажата кнопка Button1');
end;
```

Для заполнения списка можно использовать методы `AddStrings` и `AddObject`. Метод `AddStrings` позволяет при вызове увеличить содержимое списка более чем на один элемент.

Процедура `AddStrings(Strings: TStrings)` *добавляет в конец списка группу строк*, определяемую параметром `Strings`. Класс `TStrings` позволяет хранить строки и ссылки на объекты произвольного типа.

Функция `AddObject(const S: String; AObject: TObject): Integer` *добавляет в конец списка строку* `S` и связанную с ней ссылку на объект, указываемую параметром `AObject`.

Рассмотрим следующий пример заполнения списка:

```
var TS: TStringList;
...
procedure TForm1.FormCreate(Sender: TObject);
var n: integer;
begin
  TS := TStringList.Create;
  for n := 0 to PageControl1.PageCount – 1 do
    TS.AddObject(PageControl1.Pages[n].Name, PageControl1.Pages[n]);
end;
```

Здесь список `TS` заполняется перечнем названий страниц блокнота `PageControl1` при создании формы `Form1`. Вместе с названиями запоминаются ссылки на страницы. Страницы блокнота имеют тип `TTabSheet`. Знание типа необходимо при последующей работе с задаваемыми посредством ссылок объектами, в частности, для корректного выполнения операции, такой как программное переключение страниц управляющего элемента `PageControl1`. Вместо переменной `TS` можно использовать другой список подходящего типа, например, список компонента `ListBox`, доступный через свойство `Items`.

В процессе создания приложений иногда необходимо, чтобы один список содержал те же данные, что и другой. Такое согласование списков достаточно просто выполняется с помощью методов `AddStrings` и `Assign`. Последний из методов также позволяет при вызове увеличить содержимое списка более чем на один элемент. Проверить, требуется операция согласования списков или нет, можно с помощью метода `Equals`.

Процедура `Assign(Source: TPersistent)` *присваивает один объект другому*, при этом объекты должны иметь совместимые типы. В результате выполнения процедуры информация копируется из одного списка в другой с заменой содержимого. Если размеры списков (число элементов) не совпадают, то после замены число элементов заменяемого списка становится равным числу элементов копируемого списка.

Функция `Equals(Strings: TStrings): Boolean` используется для определения, *содержат ли два списка строк одинаковый текст*. Если содержимое списков совпадает, то функция возвращает значение `True`, в противном случае — значение `False`. Содержимое списков одинаково, если списки равны по длине и совпадают все их соответствующие элементы.

В приведенном далее примере производится согласование двух списков по содержанию:

```
if not ListBox2.Items.Equals(ListBox1.Items)) then begin
  ListBox2.Clear;
  ListBox2.Items.AddStrings(ListBox1.Items);
end;
```

или

```
if not ListBox2.Items.Equals(ListBox1.Items) then
  ListBox2.Items.Assign(ListBox1.Items);
```

В случае, если списки не совпадают, содержимое списка `ListBox1` копируется в список `ListBox2`, в результате чего содержимое списков становится одинаковым.

Для *удаления элементов списка* используются методы `Delete` и `Clear`. Метод `Delete(Index: Integer)` удаляет элемент с номером, заданным параметром `Index`. При попытке удаления несуществующей строки сообщение об ошибке не выдается, но метод `Delete` не срабатывает.

Например, в процедуре

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  ComboBox1.Items.Delete(4);
end;
```

при нажатии кнопки `Button2` из комбинированного списка `ComboBox1` удаляется пятая строка.

Метод `Clear` *очищает список*, удаляя все его элементы. Так, в процедуре

```
procedure TForm1.btnClearPersonalListClearClick(Sender: TObject);
begin
  lbPersonal.Items.Clear;
end;
```

при нажатии кнопки `btnClearPersonalList` очищается список `lbPersonal`.

Процедура `Move(CurIndex, NewIndex: Integer)` *перемещает элемент* из позиции с номером `CurIndex` в новую позицию с номером `NewIndex`. Если указанный номер выходит за пределы списка, то возникает исключение.

Поиск элемента в списке можно выполнить с помощью метода `IndexOf`. Процедура `IndexOf(const S: string): Integer` определяет, содержится ли строка `S` в списке. В случае успешного поиска процедура возвращает номер позиции найденной строки в списке; если строковый элемент не найден, то возвращается значение `-1`.

У класса `TStrings` есть методы `SaveToFile` и `LoadFromFile`, позволяющие непосредственно работать с текстовыми файлами. Эти методы предоставляют возможность *сохранения строк списка в текстовом файле* на диске и последующего чтения строк из этого файла. Символы файла кодируются в системе ANSI.

Процедура `SaveToFile(const FileName: string)` сохраняет строковые элементы списка в файле `FileName`. Если заданный файл отсутствует на диске, то он создается. В последующем сохраненные строки можно извлечь из файла, используя метод `LoadFromFile`. Например:

```
ListBox3.SaveToFile('C:\COMPANY\names.txt');
```

Здесь содержимое списка `ListBox3` записывается в файл names.txt каталога C:\COMPANY.

Процедура `LoadFromFile(const FileName: string)` заполняет список содержимым указанного текстового файла, при этом предыдущее содержимое списка стирается. Если заданный файл отсутствует на диске, то возникает исключение.

Пример заполнения списка содержимым файла:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ComboBox2.Items.LoadFromFile('C:\TEXT\personal.txt');
end;
```

Файл personal.txt содержит фамилии сотрудников организации. При запуске приложения содержимое этого файла загружается в комбинированный список `ComboBox2`.

При конструировании приложения изменение списка строк выполняется с помощью строкового редактора String List Editor (рис. 3.5). Его можно вызвать из окна Инспектора объектов двойным щелчком мыши в области значения свойства типа `TStrings` (например, в области значения свойства `Items` списка `ListBox`).

Рис 3.5. Строковый редактор String List Editor

Строковый редактор позволяет добавлять строки в список, удалять их из списка и изменять содержимое имеющихся строк.

Отображение текста

Текст представляет собой надпись (ярлык) и чаще всего используется в качестве заголовков для других элементов управления, которые не имеют своего свойства `Caption`. Наиболее часто для отображения надписей используется компонент `Label`, называемый также *меткой*. Он представляет собой *простой текст*, который пользователь не может отредактировать при выполнении программы.

Для управления автоматической *коррекцией размеров* компонента `Label` в зависимости от текста надписи служит свойство `AutoSize` типа `Boolean`. Если свойство имеет значение `True` (по умолчанию), то компонент `Label` изменяет свои размеры соответственно содержащемуся в нем тексту, заданному в свойстве `Caption`.

Способ *выравнивания текста* внутри компонента `Label` задает свойство `Alignment` типа `TAlignment`, которое может принимать одно из следующих значений:

- `taLeftJustify` (выравнивание по левому краю);
- `taCenter` (выравнивание по центру);
- `taRightJustify` (выравнивание по правому краю).

Замечание

Если свойство `AutoSize` имеет значение `True`, то свойство `Alignment` не действует.

Управлять *автоматическим переносом слов*, не умещающихся по ширине, на другую строку можно с помощью свойства `WordWrap` типа `Boolean`. Для длинных заголовков рекомендуется устанавливать это свойство в значение `True`, чтобы обеспечить обрезку лишних слов по ширине компонента `Label` и перенос их на следующую строку или строки. По умолчанию свойство `WordWrap` имеет значение `False`, и перенос слов заголовка не происходит.

Замечание

Свойство `WordWrap` не действует, если свойство `AutoSize` имеет значение `True`.

Надпись может быть *прозрачной* или *залитой* цветом, что определяется свойством `Transparent` типа `Boolean`. *Цвет заливки* устанавливается свойством `Color`. По умолчанию свойство `Transparent` имеет значение `False` и надпись непрозрачна. Прозрачный компонент `Label` может понадобиться в случаях, когда надпись размещается поверх рисунка и не должна закрывать изображение, например, на географической карте.

При использовании надписи в качестве заголовка для другого элемента управления желательно установить между надписью и этим элементом *ассоциированную связь*. Отметим, что компонент `Label` является неоконным элементом и получить фокус не может, однако при его выборе с помощью комбинации клавиш фокус может быть передан ассоциированному с ним элементу управления. Свойство `FocusControl` типа `TWinControl` надписи указывает на ассоциированный с ней элемент управления. Если ассоциированный с надписью элемент управления в некоторый момент времени не может получить фокус, например, когда его свойство `Enabled` имеет значение `False`, то при попытке выбора надписи Windows выдает предупреждающий сигнал, и ассоциированный элемент управления не активизируется.

Например, если компонент `Label1` является заголовком поля редактирования `Edit1`, то для свойства `FocusControl` логично установить значение следующим образом:

```
Label1.FocusControl := Edit1;
```

Напомним, что при определении заголовка надписи символ в комбинации клавиш <Alt>+<символ> указывается знаком `&` перед выбранным символом клавиатуры. При этом у компонента `Label` имеется свойство `ShowAccelChar` типа `Boolean`, определяющее, как в заголовке *интерпретируется символ* `&`. Если это свойство имеет значение `True` (по умолчанию), то амперсанд указывает "горячую" клавишу; если `False`, то амперсанд отображается как обычный символ (`&`). В этом случае связь между надписью и ассоциированным с ней элементом управления не работает, независимо от значения свойства `FocusControl`.

Замечание

Если свойство `ShowAccelChar` имеет значение `True`, то амперсанд все же можно отобразить в названии надписи как обычный символ: для этого нужно указать его дважды, например, `'&&Open'`.

Ассоциированная связь между надписью и элементом управления работает только при выборе надписи с помощью "горячей" клавиши. Чтобы элемент управления получал фокус и в случае выбора компонента `Label` с помощью мыши, можно подготовить об-

работчик события OnClick надписи. Внутри обработчика производится установка фокуса на ассоциированный с надписью элемент управления.

Приведем соответствующий пример:

```
procedure TForm1.Label1Click(Sender: TObject);
begin
  if Edit1.CanFocus then Edit1.SetFocus;
end;
```

Щелчок на надписи Label1 приводит к получению фокуса редактором Edit1. Предварительно проверяется возможность получения фокуса редактором.

Замечание

Для отображения надписей на элементе управления типа "редактор" в Delphi имеется также компонент LabeledEdit, по существу представляющий редактор (Edit) со встроенной ассоциативной связью с надписью (Label).

Так как надпись служит для отображения нередактируемого текста, иногда Label называют также *статическим текстом*. В принципе такое название соответствует назначению этого компонента, однако необходимо учитывать, что есть еще один компонент с именем — StaticText. По функциональному назначению Label и StaticText практически не отличаются. Однако компонент StaticText является наследником класса TwinControl (имеет ссылку на окно — свойство Handle) и может быть связан с другим оконным компонентом, созданным на базе класса TWinControl, например, со страницей свойств компонента ActiveX. Кроме того, компонент StaticText может отображаться в рамке, вид которой определяется его свойством BorderStyle типа TStaticBorderStyle.

Для отображения нередактируемого текста используются и другие компоненты, в частности, Edit (если его свойству ReadOnly задать значение True). Например:

```
Edit1.ReadOnly    := True;
Edit1.Color       := clBtnFace;
Edit1.Ctl3D       := False;
Edit1.BorderStyle := bsNone;
Edit1.Text        := 'Отображение текста';
```

Отображаемый компонентом Edit1 текст не отличается от текста, выводимого с использованием компонентов Label или StaticText.

Ввод и редактирование текста

Ввод и редактирование текста выполняется в специальных полях или областях формы. При необходимости пользователь может изменить отображаемые данные. Для этих целей система Delphi предлагает различные компоненты, например, Edit, MaskEdit, Memo, RichEdit и LabeledEdit.

Однострочные редакторы

Однострочный редактор, или поле редактирования, представляет собой поле ввода текста, в котором возможно отображение и изменение текста. В Delphi есть несколько однострочных редакторов, из них наиболее часто используется компонент Edit.

Компонент `Edit` позволяет *вводить* и *редактировать* с клавиатуры различные символы, при этом поддерживаются операции, такие как перемещение по строке с использованием клавиш управления курсором, удаление символов с помощью клавиш <Backspace> и <Delete>, выделение части текста и др. Отметим, что у однострочного редактора отсутствует реакция на управляющие клавиши <Enter> и <Esc>.

Для *изменения регистра* символов в поле редактирования служит свойство `CharCase` типа `TEditCharCase`, которое может принимать одно из трех значений:

- `ecLowerCase` — текст преобразуется к нижнему регистру;
- `ecNormal` — регистр символов не изменяется (по умолчанию);
- `ecUpperCase` — текст преобразуется к верхнему регистру.

При использовании компонента `Edit` для *ввода пароля* можно воспользоваться свойством `PasswordChar` типа `Char`, задающим символ для отображения в поле ввода. Этот символ при вводе текста появляется вместо фактически введенного символа. Например, после выполнения инструкций

```
Edit1.PasswordChar := '*';
Edit1.Text := 'Пароль';
```

в поле редактирования появится строка `******`, в то время как в действительности свойство `Text` имеет значение `Пароль`.

По умолчанию свойство `PasswordChar` имеет значение `#0`, и в поле редактирования отображается реально введенный текст.

Компонент `MaskEdit` также является однострочным редактором, но по сравнению с компонентом `Edit` он предоставляет дополнительную возможность ввода информации по шаблону. С помощью шаблона (маски) можно ограничить число вводимых пользователем символов, тип вводимых символов (алфавитный, цифровой и т. д.). Кроме того, во вводимую информацию можно вставить дополнительные символы (разделители при вводе даты, времени и т. п.). С помощью редактирования по маске удобно вводить телефонные номера, даты, почтовые индексы и другую информацию заранее определенного формата.

Маска задается в свойстве `EditMask` типа `String` и представляет собой последовательность специальных кодов, определяющих для поля редактирования формат содержащегося в ней текста.

Маска состоит из разделенных символом `;` трех полей:

- первое поле является собственно маской;
- второе поле — это символ, определяющий, считаются ли литеральные символы частью данных. По умолчанию используется `1`, и литеральные символы маски являются частью не только значения, редактируемого в визуальном компоненте, но и значения, содержащегося в поле. Если вместо `1` установить символ `0`, то литеральные символы маски по-прежнему будут отображаться при редактировании значения, однако в поле не сохраняются. Литеральные символы удобно применять в качестве разделителей, например, при вводе телефонных номеров;
- третье поле содержит символ, используемый для указания незаполненных символов во вводимом тексте, по умолчанию это символ подчеркивания.

Если второе и/или третье поля не определены, то для них действуют значения по умолчанию.

В маске могут использоваться следующие специальные символы:

- `!` — в тексте подавляются начальные пробелы; если символ `!` отсутствует, то подавляются конечные пробелы;
- `>` — символы, следующие за этим символом, до появления символа `<` переводятся в верхний регистр;
- `<` — символы, следующие за этим символом, до появления символа `>` переводятся в нижний регистр;
- `<>` — проверка регистра символов не производится;
- `\` — символ, следующий за этим символом, является литеральным;
- `L` — в позиции должен быть введен алфавитный символ;
- `l` (строчная латинская буква `L`) — в позиции может быть введен алфавитный символ;
- `A` — в позиции должен быть введен алфавитно-цифровой символ;
- `a` — в позиции может быть введен алфавитно-цифровой символ;
- `C` — в позиции должен быть введен символ;
- `c` — в позиции может быть введен символ;
- `0` — в позиции должен быть введен цифровой символ;
- `9` — в позиции может быть введен цифровой символ;
- `#` — в позиции может быть введен цифровой символ или знаки `+` и `-`;
- `:` — используется для разделения часов, минут и секунд в показаниях времени;
- `/` — используется для разделения дней, месяцев и лет в датах;
- `;` — разделяет поля маски;
- `_` — оставляет в окне редактирования пустое пространство, являющееся автоматически пропускаемым разделителем, в которое нельзя ввести информацию.

Примеры масок:

- `!99/99/00;1;_` — для даты;
- `!000-00-00;1;_` — для семизначного телефонного номера, разделитель "-" является частью редактируемого и запоминаемого значения;
- `!000-00-00;0;_` — для семизначного телефонного номера, разделитель "-" является частью только редактируемого значения;
- `!90:00;1;_` — для времени;
- `l_l_l_l_l_l_;1;_` — для слова длиной в шесть букв максимум (между буквами отображаются пробелы).

Для составления маски можно использовать редактор шаблонов (Input Mask Editor) (рис. 3.6). Редактор шаблонов вызывается двойным щелчком мыши в поле значения свойства `EditMask` или командой **Input Mask Editor** контекстного меню.

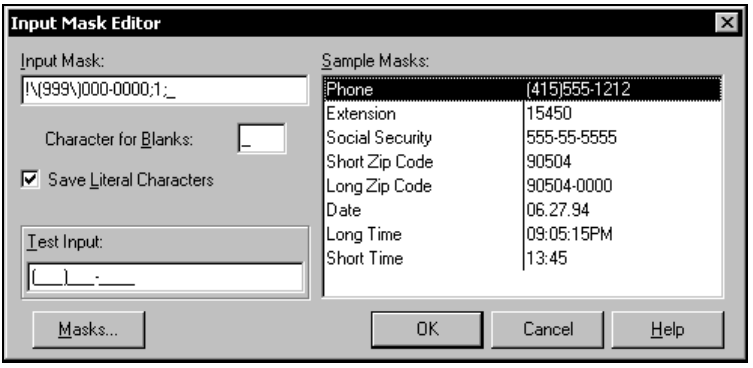


Рис. 3.6. Окно редактора шаблонов

Маска вводится в поле **Input Mask**, ее также можно выбрать из образцов в поле **Sample Masks** и затем изменить. Флажок **Save Literal Characters** управляет включением в маску литеральных символов, а поле **Character for Blanks** содержит символ, используемый для указания незаполненных позиций во входной строке. Поле **Test Input** позволяет проверить функционирование подготовленной маски. Маску также можно загрузить из файла с расширением dem, который выбирается в окне **Open Mask File**, открываемом нажатием кнопки **Masks**.

Подготовить маску можно и самому, но это более сложно. Удобно, воспользовавшись редактором маски, выбрать наиболее близкую к требуемой маску, а затем уже откорректировать ее вручную.

Компонент `LabeledEdit` представляет собой однострочный редактор с надписью и, в отличие от обычного однострочного редактора `Edit`, дополнительно имеет три свойства, управляющие надписью: `EditLabel`, `LabelPosition` и `LabelSpacing`.

Свойство `EditLabel` типа `TBoundLabel` указывает объект надписи, основные свойства которого (`Caption`, `Alignment`, `AutoSize`, `Transparent`, `Color`, `Font`) не отличаются от свойств рассмотренной ранее надписи `Label`.

Свойство `LabelPosition` типа `TLabelPosition` задает расположение надписи относительно поля редактирования и может принимать значения:

- `lpAbove` (над полем) — по умолчанию;
- `lpBelow` (под полем);
- `lpLeft` (слева от поля);
- `lpRight` (справа от поля).

Свойство `LabelSpacing` типа `Integer` указывает расстояние (в пикселах) между надписью и полем редактирования (по умолчанию 3).

Для *проверки информации*, вводимой в редакторы, можно использовать обработчики событий нажатия клавиш, например обработчик события `OnKeyPress`.

В следующем примере для редактора `Edit1` установлено разрешение ввода только десятичных цифр:

```
procedure TForm1.Edit1KeyPress(Sender :TObject; var Key :Char);
begin
  if (Key < '0') or (Key > '9') then Key := #0;
end;
```

В поле редактирования может содержаться одна строка без символа конца строки, поэтому при нажатии клавиши <Enter> не выполняются никакие действия и в строку ничего не вводится. При необходимости программист должен сам написать код действий, связанных с нажатием клавиши <Enter>. Чаще всего нажатие этой клавиши служит признаком окончания ввода информации в поле редактора, после чего следует перейти к другому элементу управления, т. е. передать ему фокус ввода, например, с помощью метода `SetFocus` или установки значения свойства `ActiveControl`.

Задать реакцию однострочного редактора на нажатие клавиши <Enter> можно так:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then begin
    Key := #0;
    Form1.ActiveControl := Edit2;
  end;
end;

procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then begin
    Key := #0;
    Edit3.SetFocus;
  end;
end;

procedure TForm1.Edit3KeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then Key := #0;
end;
```

Информация последовательно вводится в три поля, являющихся компонентами `Edit1`, `Edit2` и `Edit3`. При окончании ввода в первое или второе поля нажатие клавиши <Enter> автоматически активизирует очередное поле. Из третьего поля фокус ввода автоматически не передается. Передача фокуса ввода из разных полей реализована двумя способами: с использованием свойства `ActiveControl` формы и с помощью метода `SetFocus`.

Часто при окончании ввода в элемент редактирования и переходе к следующему (в порядке табуляции) элементу управления удобнее использовать *разделяемый (общий) обработчик события*, связанного с нажатием клавиши <Enter>. Эта процедура должна быть общей для всех элементов редактирования и может выглядеть так:

```
procedure TForm1.AllEditsKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then begin
    Form1.SelectNext(Sender as TWinControl, True, True);
```

```
    Key := #0;
  end;
end;
```

При нажатии клавиши <Enter> выполняется метод `SelectNext`, *передающий фокус ввода* следующему элементу управления. Процедура `SelectNext(CurControl: TWinControl; GoForward, CheckTabStop: Boolean)` имеет три параметра, из которых `CurControl` указывает оконный элемент управления, относительно которого выполняется передача фокуса. Параметр `GoForward` определяет направление передачи фокуса. Если его значение равно `True`, то фокус получает следующий элемент управления, в противном случае — предыдущий элемент управления. Параметр `CheckTabStop` определяет, нужно ли учитывать значение свойства `TabStop` элемента управления, который должен получить фокус. При значении `True` параметра элемент управления получит фокус, если его свойство `TabStop` также имеет значение `True`.

Чтобы приведенная процедура вызывалась в качестве обработчика для всех трех редакторов, ее нужно включить в описание класса формы и указать в качестве обработчика события `OnKeyPress`:

```
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    procedure AllEditsKeyPress(Sender: TObject; var Key: Char);
    procedure FormCreate(Sender: TObject);
  end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Edit1.OnKeyPress := AllEditsKeyPress;
  Edit2.OnKeyPress := AllEditsKeyPress;
  Edit3.OnKeyPress := AllEditsKeyPress;
end;
```

Аналогично можно создать разделяемую процедуру, общую для нескольких компонентов (в том числе и разных, например, `Edit` и `Memo`), выполняющую обработку других событий.

В качестве еще одного варианта обработки нажатия клавиши <Enter> можно использовать процедуру обработки события `OnKeyPress` самой формы, на которой находятся элементы редактирования. Свойству `KeyPreview` формы устанавливается значение `True`, чтобы обработчик формы получал сообщение о нажатии клавиш первым. Процедура-обработчик может быть такой:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then begin
    Form1.SelectNext(Form1.ActiveControl as TWinControl, True, True);
    Key := #0;
  end;
end;
```

Аналогичным образом выполняется обработка нажатия клавиши <Esc> и других клавиш.

Многострочный редактор

Для работы *с многострочным текстом* Delphi предоставляет компонент `Memo`. Многострочный редактор имеет практически те же возможности по редактированию текста, что и однострочные редакторы. Главное отличие этих элементов управления заключается в том, что многострочный редактор содержит несколько строк текста.

Для *доступа ко всему содержимому* многострочного редактора используется свойство `Text` типа `String`. В этом случае все содержимое компонента `Memo` представляется одной строкой, а конец строки, вставляемый при нажатии клавиши <Enter>, записывается двумя кодами `#13#10`, поэтому видимых пользователем символов будет меньше, чем их содержится в строке `Text`. Эту особенность нужно учитывать, например, при определении позиции заданного символа в какой-либо строке компонента `Memo`.

Для работы с *отдельными строками* используется свойство `Lines` типа `TStrings`. Класс `TStrings` служит для выполнения операций со строками и имеет различные свойства и методы, которые уже были рассмотрены выше. Компонент `Memo` позволяет использовать возможности этого класса через свое свойство `Lines`.

Рассмотрим примеры операций с многострочным редактором:

```
Memo1.Lines[3] := 'asd';
Memo2.Lines.Clear;
Memo3.Lines.Add('Новая строка');
```

Здесь четвертой строке редактора `Memo1` присваивается новое значение `asd` (напомним, что в классе `TStrings` нумерация строк начинается с нуля). Содержимое редактора `Memo2` полностью очищается. В конец текста редактора `Memo3` добавляется новая строка.

Содержимое компонента `Memo` можно *загружать* из текстового файла и *сохранять* в текстовом файле. Для этого удобно использовать методы `LoadFromFile(const FileName: String)` и `SaveToFile(const FileName: String)` класса `TString`. Параметр `FileName` методов определяет текстовый файл для операций чтения и записи.

Пример чтения информации из текстового файла в компонент `Memo1`:

```
Memo1.Lines.LoadFromFile('D:\Delphi7\Chapter3\example1.txt');
```

и записи информации из компонента `Memo2` в текстовый файл:

```
Memo2.Lines.SaveToFile('D:\Delphi7\Chapter3\example2.txt');
```

Для удобного *просмотра информации* можно задать в поле редактирования полосы прокрутки с помощью свойства `ScrollBars` типа `TScrollStyle`, принимающего следующие значения:

- `ssNone` (полосы прокрутки отсутствуют) — по умолчанию;
- `ssHorizontal` (горизонтальная полоса прокрутки снизу);
- `ssVertical` (вертикальная полоса прокрутки справа);
- `ssBoth` (есть обе полосы прокрутки).

Текст в поле компонента `Memo` может быть выровнен различными способами. Способ *выравнивания* определяет свойство `Alignment` типа `TAlignment`, которое может принимать одно из следующих значений:

- `taLeftJustify` (выравнивание по левой границе) — по умолчанию;
- `taCenter` (выравнивание по центру);
- `taRightJustify` (выравнивание по правой границе).

В отличие от однострочного редактора, компонент `Memo` обладает возможностью реакции на *нажатие клавиши <Enter>*. Чтобы при этом происходил ввод новой строки, свойству `WantReturns` типа `Boolean` должно быть установлено значение `True` (по умолчанию). В противном случае редактор не реагирует на нажатие клавиши <Enter>.

Аналогично значение свойства `WantTabs` типа `Boolean` определяет реакцию компонента на *нажатие клавиши <Tab>*. Если свойство установлено в значение `True`, то при нажатии клавиши <Tab> в текст вставляются символы табуляции. По умолчанию свойство `WantTabs` имеет значение `False`, и при нажатии клавиши <Tab> редактор передает фокус ввода следующему оконному элементу управления.

Интервал табуляции задается при создании компонента `Memo`, по умолчанию он равен 32 единицам. Для изменения интервала нужно послать редактору сообщение `EM_SetTabStops`, передав в качестве второго параметра ссылку на константу, содержащую массив или одиночное значение целочисленного типа. Первый параметр сообщения задает количество значений, содержащихся во втором параметре. Значения задаются в единицах измерения диалога.

Рассмотрим на примере, как производится установка значений табуляторов:

```
procedure TForm1.btnTabClick(Sender: TObject);
const PosTab1: integer = 100;
      PosTab2: array[0..2] of integer = (15, 20, 25);
begin
  Memo1.Perform(em_SetTabStops, 1, Longint(@PosTab1));
  Memo2.Perform(em_SetTabStops, 3, Longint(@PosTab2));
end;
```

Здесь для компонента `Memo1` определяется новое значение табулятора, равное 100 единицам. Для компонента `Memo2` устанавливаются три значения табулятора.

Замечание

Даже в случае, когда свойства `WantReturns` и `WantTabs` имеют значение `False`, компонент `Memo` способен обработать нажатия клавиш <Enter> или <Tab> при нажатой клавише <Ctrl>.

Компонент `RichEdit` является компонентом редактирования *с форматированием* текста и в дополнение к `Memo` поддерживает такие операции форматирования, как выравнивание и табуляция текста, применение отступов, изменение гарнитуры и др. Текст, содержащийся в этом элементе редактирования, совместим с форматом RTF (Rich Text Format), поддерживаемым многими текстовыми процессорами в среде Windows.

Общие элементы компонентов редактирования

Компоненты, используемые для редактирования информации, похожи друг на друга и, соответственно, имеют много общих свойств, событий и методов.

При любых *изменениях в содержимом* редактора возникает событие `OnChange` типа `TNotifyEvent`, которое можно использовать для проверки текста, содержащегося в поле

ввода, например, для оперативного контроля правильности набора данных. Кроме того, *при изменении содержимого* редактора свойство `Modified` типа `Boolean` принимает значение `True`. Это свойство можно использовать, в частности, для проверки того, сохранена ли редактируемая информация на диске:

```
if Memo1.Modified then begin
   // Инструкции выдачи предупреждения и сохранения информации
end;
```

Для *указания максимального количества символов*, которые допускается вводить в поле редактирования, можно использовать свойство `MaxLength` типа `Integer`. При этом ограничение на длину текста относится к вводу со стороны пользователя, программно можно ввести количество символов большее, чем задано в свойстве `MaxLength`. По умолчанию длина вводимого пользователем текста не ограничена (`MaxLength = 0`).

Свойства `AutoSelect`, `SelStart`, `SelLength` и `SelText` позволяют работать с *выделенным фрагментом* текста. Эти свойства доступны не только для чтения, например, в случае анализа текста, выделенного пользователем, но и для записи, когда фрагмент выделяется программно, скажем, в процессе поиска или замены текста.

Свойство `AutoSelect` типа `Boolean` определяет, будет ли *автоматически выделен текст* в поле редактирования, при получении последним фокуса ввода (по умолчанию имеет значение `True`).

Значение свойства `SelText` типа `String` определяет *выделенный фрагмент*. При отсутствии выделенного текста значением свойства является пустая строка.

Свойства `SelStart` и `SelLength` типа `Integer` задают начальную *позицию* в строке (отсчет символов в строке начинается с нуля) и *длину* выделенного фрагмента соответственно.

Замечание

Свойства `SelStart` и `SelLength` взаимозависимы, поэтому при выделении фрагмента программным способом сначала необходимо установить значение свойства `SelStart`, а затем определять длину выделенного текста, задавая значение свойства `SelLength`.

Если фрагмент выделяется программно, например, в случае поиска строки, и должен быть *выделен цветом*, то свойству `HideSelection` типа `Boolean` следует установить значение `False`. Это свойство определяет, будет ли отображаться выделенный текст при потере компонентом фокуса ввода.

Если свойство `HideSelection` имеет значение `True` (по умолчанию), то текст не будет выглядеть выделенным при переходе фокуса на другой элемент управления.

Рассмотрим несколько операций с выделенным текстом:

```
Memo1.SelStart  := 19;
Memo1.SelLength := 6;
Memo1.SelText   := 'abcdefgh';
if pos('qwerty', Edit1.Text) <> 0 then begin
   Edit1.HideSelection := False;
   Edit1.SelStart      := pos('qwerty', Edit1.Text) - 1;
   Edit1.SelLength     := length('qwerty');
end;
```

В компоненте `Memo1` 6 символов, начиная с 19-го, заменяются строкой `abcdefgh`. В компоненте `Edit1` осуществляется поиск строки `qwerty`. В случае удачного поиска найденный фрагмент выделяется.

Кроме свойств, для *операций с выделенным фрагментом* текста используются также методы `SelectAll`, `CopyToClipBoard` и `CutToClipBoard`.

Метод `SelectAll` *выделяет весь текст* в элементе редактирования.

Метод `CopyToClipBoard` *копирует*, а `CutToClipBoard` *вырезает* в буфер обмена выделенный фрагмент текста. Например, инструкция `Memo1.CutToClipBoard;` вырезает выделенный фрагмент и помещает его в буфер обмена.

Для работы с *буфером обмена* имеется также метод `PasteFromClipBoard`, вставляющий текст из буфера обмена в текущую позицию курсора в поле редактирования. Если имеется выделенный фрагмент, то вставляемый текст заменяет его. Более подробно вопросы, связанные с буфером обмена, рассматриваются в *главе 37*, посвященной обмену информацией.

Для *проверки текста*, введенного в поле редактирования, можно использовать событие `OnExit`, возникающее при окончании ввода, т. е. при потере этим элементом фокуса ввода. В качестве примера рассмотрим следующую процедуру:

```
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  if (Edit1.Text = '123') or (Edit1.Text = '456') then begin
     MessageDlg('Артикул ' + Edit1.Text + ' неправильный!' +
                 #13#10'Повторите ввод.', mtError, [mbOK], 0);
     Edit1.SetFocus;
     Edit1.SelectAll;
  end;
end;
```

Здесь для нового товара в редактор `Edit1` вводится артикул, который не должен быть равен `123` или `456` (в реальных приложениях проверку реализовать сложнее, т. к. артикул должен отвечать более сложным требованиям, например, быть уникальным и отличаться от уже имеющихся). При окончании ввода в обработчике события `OnExit` выполняется проверка артикула. Если он набран неверно, то выдается предупреждение, а редактор `Edit1` снова получает фокус.

Основное назначение полей редактирования — ввод и изменение текста, однако их можно использовать и для *отображения нередактируемого текста*, например, при выводе справочной информации. С этой целью нужно установить соответствующие значения свойств `ReadOnly` или `Enabled`. Оба способа обеспечивают отображение нередактируемого текста, однако имеют свои недостатки.

В случае использования свойства `ReadOnly`, например, следующим образом:

```
Memo1.ReadOnly  := True;
Memo1.Alignment := taCenter;
Memo1.Clear;
Memo1.Lines.Add('Пример');
Memo1.Lines.Add('справочной');
Memo1.Lines.Add('информации');
```

компонент Memo при выполнении программы может получать фокус. Здесь выполняется очистка поля ввода и добавление трех отдельных строк текста, выровненных по центру. При этом в поле ввода отображается курсор, который можно перемещать, что создает у пользователя иллюзию доступности текста для редактирования.

Курсор не отображается, если свойство Enabled используется так:

```
Memo1.Enabled   := False;
Memo1.Alignment := taCenter;
Memo1.Clear;
Memo1.Lines.Add('Пример');
Memo1.Lines.Add('справочной');
Memo1.Lines.Add('информации');
```

Однако в этом случае поле редактирования становится неактивным, и находящийся в нем текст отображается бледным цветом, что не слишком удобно для чтения. Кроме того, отключаются полосы прокрутки (при их наличии).

Поэтому на практике для отображения нередактируемого текста чаще используется свойство ReadOnly.

Работа со списками

Список представляет собой упорядоченную совокупность взаимосвязанных элементов, являющихся текстовыми строками. Списки широко применяются в Windows, например, для отображения информации о перечне шрифтов или способах их начертания (рис. 3.7). Элементы списка могут быть отсортированы в алфавитном порядке или размещены в порядке их добавления в список. Как и другие объекты, представляющие собой совокупность данных, списки позволяют добавлять, удалять и выбирать отдельные их элементы (строки).

Простой список

Простой список представляет собой прямоугольную область, в которой располагаются его строковые элементы. Для работы с простым списком в Delphi предназначен компонент ListBox.

Если количество строк больше, чем их может поместиться в видимой области списка, то у области отображения появляется полоса прокрутки. *Ориентация* полосы прокрутки, а также *число столбцов*, которые одновременно видны в области списка, зависят от свойства Columns типа Integer. По умолчанию свойство имеет значение 0. Это означает, что все элементы списка расположены в одном столбце, и при необходимости автоматически появляется (рис. 3.8, список справа) или исчезает (рис. 3.8, список слева) вертикальная полоса прокрутки.

Если свойство Columns имеет значение, большее или равное 1, то в области списка всегда присутствует горизонтальная полоса прокрутки, а элементы разбиваются на такое число столбцов, чтобы можно было, прокручивая список по горизонтали, просмотреть все его элементы. При этом в видимой области списка отображается число столбцов, равное значению свойства Columns. Так, на рис. 3.9 приведены два списка, содержащие

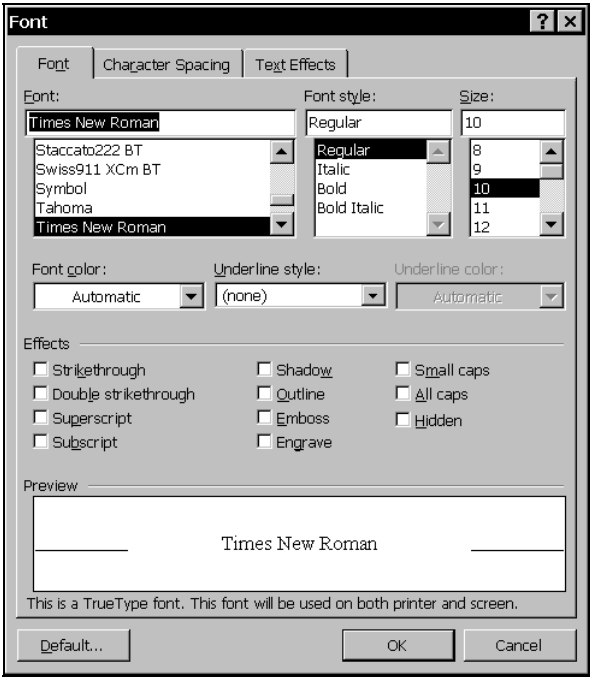


Рис. 3.7. Списки в диалоговом окне **Font**

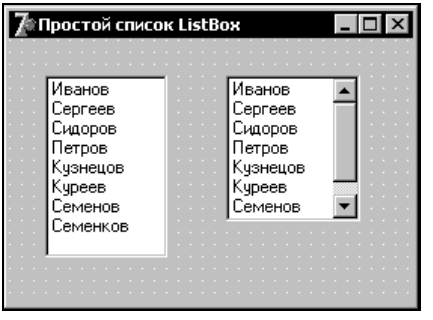


Рис. 3.8. Варианты списков

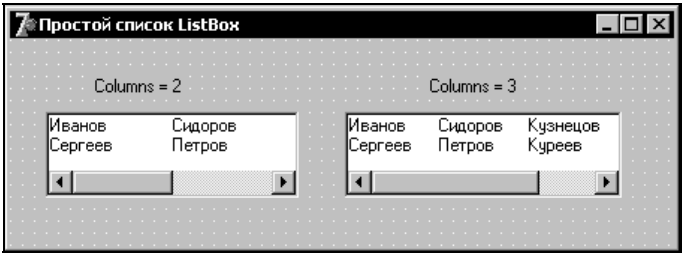


Рис. 3.9. Списки с горизонтальной полосой прокрутки

по 9 фамилий каждый. Для левого списка значение свойства `Columns` равно 2, и в списке одновременно видны 2 столбца, для правого списка свойство `Columns` имеет значение 3, поэтому отображены все 3 столбца.

Иногда требуется, чтобы в списке одновременно отображались и вертикальная, и горизонтальная полосы прокрутки. В этом случае нужно задать свойству `Columns` нулевое значение, тогда вертикальная полоса прокрутки будет появляться по мере надобности. Для отображения горизонтальной полосы прокрутки следует послать списку сообщение `LB_SetHorizontalExtent`. Третьим параметром сообщения является максимальное значение полосы прокрутки в пикселах. Если задать это значение заведомо большим, чем размер списка `ListBox`, то горизонтальная полоса прокрутки будет отображаться всегда. Четвертый параметр сообщения в этом случае равен нулю. Если горизонтальная полоса прокрутки не нужна, то можно послать еще одно сообщение, указав в качестве максимального размера значение, равное нулю.

Приведем пример списка `ListBox1` с двумя полосами прокрутки:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Columns := 0;
  SendMessage(ListBox1.Handle, LB_SetHorizontalExtent, 1000, 0);
end;
```

При работе со списком программист может *управлять номером элемента*, который в видимой области списка отображается верхним. Эта возможность обеспечивается свойством `TopIndex` типа `Integer`, доступным на этапе выполнения приложения. Так, в приведенной далее процедуре в списке `ListBox1` верхним отображается элемент, номер которого задает реверсивный счетчик `UpDown1`:

```
procedure TForm1.UpDown1Click(Sender: TObject; Button: TUDBtnType);
begin
  ListBox1.TopIndex := UpDown1.Position;
end;
```

Стиль простого списка определяет свойство `Style` типа `TListBoxStyle`, принимающее следующие значения:

- `lbStandard` (стандартный стиль) — по умолчанию;
- `lbOwnerDrawFixed` (список с элементами фиксированной высоты, устанавливаемой свойством `ItemHeight`);
- `lbOwnerDrawVariable` (список с элементами, которые могут иметь разную высоту).

Если стиль списка отличен от значения `lbStandard`, то за прорисовку элементов списка отвечает программист. Для этих целей используются графические возможности Delphi.

Список может иметь обычную рамку или не иметь рамки вообще. *Наличие рамки* определяет свойство `BorderStyle` типа `TBorderStyle`, принимающее два возможных значения:

- `bsNone` (рамки нет);
- `bsSingle` (рамка есть) — по умолчанию.

Комбинированный список

Комбинированный список объединяет поле редактирования и список. Пользователь может выбирать значение из списка или вводить его непосредственно в поле. В Delphi для работы с комбинированным списком служат компоненты `ComboBox` и `ComboBoxEx`. Компонент `ComboBox` представляет собой обычный комбинированный список, а компонент `ComboBoxEx` — расширенный комбинированный список. Рассмотрим здесь компонент `ComboBox`, а особенностям компонента `ComboBoxEx` посвятим отдельный раздел.

Отметим также, что в Delphi имеется специализированный комбинированный список `ColorBox`, предназначенный для выбора цвета, который задается через свойство `Selected` типа `TColor`.

Список, инкапсулированный в компоненте `ComboBox`, может быть простым либо раскрывающимся. Раскрывающийся список в свернутом виде занимает на экране меньше места. На рис. 3.10 показан компонент `ComboBox` со свернутым и развернутым списками.

В отличие от простого, комбинированный список не может иметь горизонтальную полосу прокрутки и допускает выбор только одного значения.

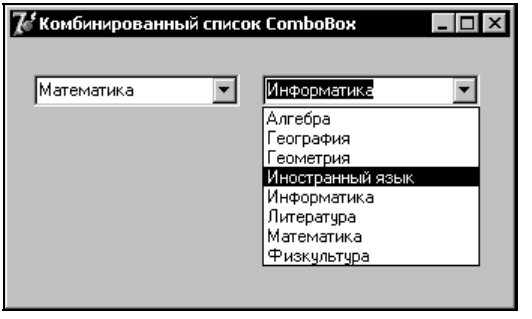


Рис. 3.10. Компонент `ComboBox`

Свойство `Style` типа `TComboBoxStyle` определяет *внешний вид* и *поведение* комбинированного списка. Свойство `Style` принимает следующие значения:

- `csDropDown` (раскрывающийся список с полем редактирования) — по умолчанию: пользователь может выбирать элементы в списке (при этом выбранный элемент появляется в поле ввода) или вводить (редактировать) информацию непосредственно в поле ввода;
- `csSimple` (поле редактирования с постоянно раскрытым списком); для того чтобы список был виден, необходимо увеличить высоту (свойство `Height`) компонента `ComboBox`;
- `csDropDownList` (раскрывающийся список, допускающий выбор элементов в списке);
- `csOwnerDrawFixed` (список с элементами фиксированной высоты, задаваемой свойством `ItemHeight`);
- `csOwnerDrawVariable` (список с элементами, которые могут иметь разную высоту).

Если стиль списка имеет значение `csOwnerDrawFixed` или `csOwnerDrawVariable`, то за прорисовку элементов списка отвечает программист.

Свойство DropDownCount типа Integer определяет *количество строк*, которые одновременно отображаются в раскрывающемся списке. Если значение свойства превышает число строк списка, определяемое значением подсвойства Count свойства Items, то у раскрывающегося списка автоматически появляется вертикальная полоса прокрутки. Если размер списка меньше, чем задано в свойстве DropDownCount, то отображаемая область списка автоматически уменьшается. Свойство DropDownCount по умолчанию имеет значение 8.

Свойство DroppedDown типа Boolean позволяет определить, *раскрыт ли список*. Если свойство имеет значение True, то список отображается в раскрытом виде, в противном случае список свернут. Свойство DroppedDown действует только, если свойство Style имеет значение, не равное csSimple. При отображении раскрывающегося списка возникает событие OnDropDown типа TNotifyEvent. Программист может самостоятельно управлять раскрытием и свертыванием списка в процессе выполнения приложения, устанавливая свойство DroppedDown в соответствующее значение.

Рассмотрим пример управления раскрытием и свертыванием списка:

```
procedure TForm1.btnOpenListClick(Sender: TObject);
begin
  ComboBox2.DroppedDown := True;
end;

procedure TForm1.btnCloseListClick(Sender: TObject);
begin
  ComboBox2.DroppedDown := False;
end;
```

Нажатие кнопки btnOpenList приводит к открытию, а кнопки btnCloseList — к свертыванию списка ComboBox2.

При работе с комбинированным списком генерируются следующие события типа TNotifyEvent:

- OnDropDown (открытие списка);
- OnCloseUp (закрытие списка);
- OnSelect (выбор элемента);
- OnChange (изменение текста в поле редактирования).

Общая характеристика списков

Простой и комбинированный списки во многом похожи друг на друга и имеют много общих свойств, методов и событий. Основным для списков является рассмотренное ранее свойство Items, которое содержит элементы списка и представляет собой коллекцию (массив) строк.

Элементы списка можно отсортировать в алфавитном порядке. Наличие или отсутствие *сортировки* определяет свойство Sorted типа Boolean. При значении False этого свойства (по умолчанию) элементы в списке располагаются в порядке их поступления в список. Если же свойство Sorted имеет значение True, то элементы автоматически сортируются по алфавиту в порядке возрастания. На рис. 3.11 показаны неотсортиро-

ванный (слева) и отсортированный (справа) списки. Если для отсортированного списка свойство `Sorted` снова установить в значение `False`, то порядок элементов списка не изменится. В этом случае значение свойства сортировки будет действовать только для новых строк, добавляемых в список.

Рис. 3.11. Сортировка списков

Для кодирования символов, включающих русские буквы, применяется вариант Windows 1251 кода ANSI. В Windows сортировка этих символов осуществляется в порядке возрастания значений кодов с учетом регистра букв. Младшими по значению считаются специальные символы и разделители, такие как точка, тире, запятая и др., затем следуют буквы латинского алфавита, при этом символы, отличающиеся только регистром, располагаются рядом: сначала — символ в нижнем регистре, затем — в верхнем регистре, несмотря на то, что их коды не являются соседними. Последними упорядочиваются буквы русского алфавита, при этом символы, отличающиеся лишь регистром, также располагаются рядом.

Действие свойства `Sorted` является статическим, а не динамическим. Это означает, что при добавлении к отсортированному списку методами `Insert` и `Add` новых строк они размещаются на указанных позициях или в конце списка, а не по алфавиту. Чтобы отсортировать список, нужно сбросить значение свойства `Sorted` в `False`, а затем снова установить значение `True`:

```
ListBox1.Sorted := False;
ListBox1.Sorted := True;
```

Пользователь может *выбирать* отдельные строки списка с помощью мыши и клавиатуры. Выбранный в списке элемент определяется свойством `ItemIndex` типа `Integer`. При анализе номеров строк нужно иметь в виду, что отсчет начинается с нуля, поэтому, например, 7-я строка имеет номер 6. В частности, отобразить номер выбранной в списке (`ListBox1`) строки можно с помощью следующей инструкции:

```
Label5.Caption := 'В списке выбрана ' +
                  IntToStr(ListBox1.ItemIndex) + ' строка';
```

Программист может выбрать элемент списка, установив в требуемое значение свойство `ItemIndex`. Так, инструкция `ListBox2.ItemIndex := 3;` приводит к выбору четвертой строки списка `ListBox2`, и это отображается на экране.

По умолчанию в списке можно выбрать один элемент. Для выбора двух и более элементов свойство `MultiSelect` типа `Boolean`, *управляющее возможностью выбора нескольких строк*, устанавливается в значение `True`. По умолчанию свойство `MultiSelect` имеет значение `False`.

Замечание

Если свойство `MultiSelect` установлено в значение `True`, то свойство `ItemIndex` содержит номер последнего элемента, на котором был выполнен щелчок. При этом не учитывается выполненное действие — выбор строки или отмена выбора. Подобная особенность может стать причиной ошибок, поэтому желательно использовать свойство `ItemIndex` только для списков, не поддерживающих множественный выбор элементов.

В случае, когда список поддерживает возможность выбора нескольких строк, свойство `ExtendedSelect` типа `Boolean` управляет *способом выбора* нескольких элементов. Когда свойство `ExtendedSelect` имеет значение `False`, добавить к выбранной группе очередной элемент можно только с помощью мыши. При этом первый щелчок мыши на строке выбирает ее, а повторный щелчок отменяет выбор строки. Если свойство `ExtendedSelect` имеет значение `True` (по умолчанию), то в дополнение к мыши можно выбирать элементы с помощью клавиш управления курсором, <Shift> и <Ctrl>.

Поскольку комбинированный список допускает одновременный выбор только одного элемента, у него нет свойств `MultiSelect` и `ExtendedSelect`.

Число выбранных элементов в списке возвращает свойство `SelCount` типа `Integer`. Для определения *номеров выбранных строк* можно просмотреть значения свойства `Selected[Index: Integer]` типа `Boolean`, представляющего собой массив логических значений. Если строка с номером `Index` выбрана, то ее признак `Selected` принимает значение `True` и наоборот, если строка не выбрана, то признак `Selected` имеет значение `False`. Свойства `SelCount` и `Selected` обычно используются для списков, поддерживающих множественный выбор элементов.

Пример операции с выбранными элементами списка:

```
var i :integer;
...
for I := 0 to ListBox2.Items.Count – 1 do
  if ListBox2.Selected[i] then ListBox2.Items[i] := 'Строка выбрана';
```

В этом примере все выбранные строки списка `ListBox2` заменяются текстом `Строка выбрана`.

Свойство `Selected` можно использовать и для программного выбора элементов, устанавливая значение `True` для тех строк, которые должны быть выбраны. Например:

```
ListBox1.Selected[1] := True;
ListBox1.Selected[3] := True;
```

При выборе элемента списка происходит событие `OnClick`, которое можно использовать для *обработки выбранных строк*. Например, так:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  Label3.Caption := ListBox1.Items[ListBox4.ItemIndex];
end;
```

Надпись `Label3` отображает элемент списка `ListBox1`, номер которого совпадает с номером элемента, выбранного в списке `ListBox4`.

Обычно список сам отображает свои элементы, однако программист может организовать и собственную *прорисовку списка*. Обычно это делается в случае, когда свойство `Style` (стиль списка) имеет значения `lbOwnerDrawFixed` или `lbOwnerDrawVariable` для простого списка и `csOwnerDrawFixed` или `csOwnerDrawVariable` — для комбинированного списка. Для кодирования *операций по отображению элементов* списка используется событие `OnDrawItem` типа `TDrawItemEvent`, возникающее при необходимости повторного отображения элемента в области списка. Тип события `OnDrawItem` описан следующим образом:

```
type TDrawItemEvent = procedure (Control: TWinControl;
     Index: Integer; Rect: TRect; State: TOwnerDrawState) of object;
```

Параметр `Control` является ссылкой на список, в котором находится подлежащий прорисовке элемент. Использование этого параметра может понадобиться в случае, когда для прорисовки нескольких списков применяется общий обработчик. Номер в списке выводимого элемента задает параметр `Index`, а параметр `Rect` определяет прорисовываемую область, занимаемую элементом. Параметр `State`, отвечающий за состояние элемента списка, может принимать комбинации значений:

- `odSelected` (элемент выделен);
- `odDisabled` (элемент заблокирован);
- `odFocused` (элемент имеет фокус);
- `odGrayed` (элемент недоступен) — для комбинированного списка;
- `odChecked` (элемент отмечен) — для комбинированного списка.

Чтобы познакомиться с тем, как на практике происходит программная прорисовка элементов списка, рассмотрим следующий пример (листинг 3.1).

Листинг 3.1. Пример программной прорисовки списков

```
unit uLBDraw;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure ListBox1DrawItem(Control: TWinControl; Index: Integer;
                               Rect: TRect; State: TOwnerDrawState);
    procedure ListBox1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var               Form1: TForm1;
    bmSelect, bmNoSelect: TBitmap;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  // Загрузка рисунков из файлов
  bmSelect := TBitmap.Create; mNoSelect := TBitmap.Create;
  bmSelect.LoadFromFile('picture1.bmp');
  bmNoSelect.LoadFromFile('picture2.bmp');

  // Включение возможности множественного выбора элементов
  ListBox1.MultiSelect := True;

  // Установка стиля, требующего программной прорисовки
  // элементов списка
  ListBox1.Style := lbOwnerDrawFixed;

  // Задание высоты элементов списка и размера шрифта
  ListBox1.Font.Height   := bmSelect.Height;
  ListBox1.ItemHeight    := ListBox1.Font.Height + 6;
  ListBox1.IntegralHeight := True;

  // Заполнение списка
  ListBox1.Items.Add('Первая строка');
  ListBox1.Items.Add('Вторая строка');
  ListBox1.Items.Add('Третья строка');
  ListBox1.Items.Add('Четвертая строка');
  ListBox1.Items.Add('Пятая строка');
end;

// Программная прорисовка элементов списка
procedure TForm1.ListBox1DrawItem(Control: TWinControl;
               Index: Integer; Rect: TRect; State: TOwnerDrawState);
var r: Trect;
begin
  if TListBox(Control).Selected[Index]
    then begin
      TListBox(Control).Canvas.Font.Color := clYellow;
      TListBox(Control).Canvas.BrushCopy(Bounds(Rect.Left + 2, Rect.Top,
               bmSelect.Width, bmSelect.Height), bmSelect,
               Bounds(0, 0, bmSelect.Width, bmSelect.Height), clRed);
     end
  else begin
     TListBox(Control).Canvas.Font.Color := clBlack;
     TListBox(Control).Canvas.BrushCopy(Bounds(Rect.Left + 2, Rect.Top,
          bmNoSelect.Width, bmNoSelect.Height), bmNoSelect,
          Bounds(0, 0, bmNoSelect.Width, bmNoSelect.Height), clBlue);
  end;
```

```
  r := Rect;
  r.Left := r.Left + bmSelect.Width + 6;
  TListBox(Control).Canvas.FillRect(r);
  TListBox(Control).Canvas.TextOut(r.Left, Rect.Top,
                (Control as TListBox).Items[Index]);
end;

// Перерисовка списка при выборе или отмене выбора элемента
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  ListBox1.Refresh;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  bmSelect.Free; bmNoSelect.Free;
end;
end.
```

В каждой строке списка `ListBox1` слева от текста содержится рисунок, показывающий, выделен данный элемент или нет. Для выделенных элементов отображается рисунок, загруженный из файла picture1.bmp, для невыделенных элементов — рисунок из файла picture2.bmp. Выделенные элементы дополнительно отмечаются желтым цветом символов на синем фоне.

Поскольку свойство `Style` списка `ListBox1` установлено в значение `lbOwnerDrawFixed`, процедура `ListBox1DrawItem` вызывается автоматически каждый раз, когда необходима прорисовка элементов списка. В этой процедуре кодируются все действия, связанные с установкой цветов, выводом рисунков и текста. Мы не будем сейчас ее разбирать, т. к. все вопросы, связанные с загрузкой текста, выбором карандаша и т. п., подробно рассматриваются далее. Перерисовка списка осуществляется в методе `Refresh`, вызываемом в обработчике события `OnClick`.

При самостоятельной прорисовке видимой области списка программно можно *управлять высотой* отдельных строк с помощью свойства `ItemHeight` типа `Integer`. Если список отображается обычным способом, то значение свойства определяется высотой шрифта, используемого для элементов списка.

Простой и комбинированный списки не имеют собственного заголовка, поэтому для их *обозначения* используются компоненты `Label` или `StaticText`. Обычно текстовая надпись располагается над списком и связана с ним таким образом, чтобы значение ее свойства `FocusControl` указывало на имя списка, например:

```
Label1.FocusControl := ComboBox2;
```

Напомним, что если надпись так ассоциирована со списком, то ее выбор с помощью комбинации <Alt>+<клавиша, заданная в свойстве `Caption`>, приводит к установке фокуса ввода на список.

Как и класс `TStrings`, список имеет одноименный метод `Clear`, удаляющий все строки списка.

Так, в процедуре

```
procedure TForm1.btnClearClick(Sender: TObject);
begin
  ListBox1.Items.Clear;
  ListBox2.Clear;
end;
```

при нажатии кнопки `btnClear` содержимое списков `ListBox1` и `ListBox2` очищается разными способами.

Особенности расширенного комбинированного списка

В отличие от обычного комбинированного списка `ComboBox`, в элементах (строках) компонента `ComboBoxEx` могут одновременно содержаться изображение (слева) и текст (справа).

Элементы компонента `ComboBoxEx` имеют тип `TListControlItem`, основными свойствами которого являются:

- `Caption` типа `String` (текст элемента);
- `ImageIndex` типа `Integer` (номер изображения в списке образов);
- `Data` типа `Pointer` (указатель на данные, определенные программистом и связанные со списком).

Выводимые в строке изображения должны быть предварительно помещены в список графических образов `ImageList` (будет рассмотрен в *главе 10*, посвященной графике). На список графических образов указывает свойство `Images` типа `TListImages`.

Элементы компонента `ComboBoxEx` содержатся в свойстве `ItemsEx` типа `TComboExItems`. В отличие от строковых элементов обычного комбинированного списка, их нельзя редактировать на этапе проектирования приложения. Добавление и удаление этих элементов выполняются динамически методами `AddItem` и `Delete`.

Функция

```
AddItem(Caption: String, ImageIndex: Integer,
        SelectedImageIndex: Integer, OverlayImageIndex: Integer,
        Indent: Integer, Data: Pointer): TComboExItem
```

добавляет к списку элемент, который содержит текст, заданный параметром `Caption`, и изображение, указанное параметром `ImageIndex`. Параметр `SelectedImageIndex` определяет изображение элемента, когда он выделен, а `Indent` — отступ изображения и текста от левого края. Параметр `Data` позволяет указать данные, с которыми связан добавляемый элемент. В качестве результата функция возвращает добавленный элемент.

Процедура `Delete(Index: Integer)` удаляет элемент с указанным номером. Нумерация элементов начинается с нуля. При попытке удалить элемент с несуществующим номером генерируется исключение.

Рассмотрим пример работы с элементами компонента ComboBoxEx:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  ComboBoxEx1.Images := ImageList1;
  ComboBoxEx1.ItemsEx.Delete(2);
  ComboBoxEx1.ItemsEx.AddItem('111', 0, 0, 0, 0, nil);
  ComboBoxEx1.ItemsEx.AddItem('222', 1, 1, 1, 0, nil);
  ComboBoxEx1.ItemsEx.AddItem('333', 2, 2, 2, 0, nil);
end;
```

Из списка компонента ComboBoxEx1 удаляется третий элемент, и к списку добавляются три новых элемента с текстом 111, 222, 333, соответственно, и изображениями с номерами 0, 1, 2, которые берутся из компонента ImageList1.

Еще одной особенностью компонента ComboBoxEx является его свойство Action типа TBasicAction. Используя это свойство, компонент ComboBoxEx можно связать с объектом действия, в результате при выборе элемента списка автоматически будет выполнен обработчик события OnExecute заданного объекта действия. (Более подробно использование объектов действия рассматривается при описании синхронизации элементов управления в *главе 5*, посвященной работе с меню.)

Менее существенным отличием расширенного комбинированного списка от обычного является то, что компонент ComboBoxEx не реагирует, например, на изменение значения свойства ItemHeight.

Пример приложения

Чтобы проиллюстрировать работу со списками, рассмотрим приложение, в котором организовано взаимодействие двух списков: ListBox1 и ListBox2 (рис. 3.12). При запуске приложения первый список заполняется названиями месяцев, а второй список очищается. Оба списка допускают множественный выбор элементов. При нажатии кнопки btnRight с заголовком -----> выбранные в первом списке элементы переходят во второй список. Кнопка btnLeft с заголовком <----- вызывает передачу выбранных элементов в обратном направлении. Оба списка поддерживают перетаскивание одиночных элементов между списками с помощью мыши.

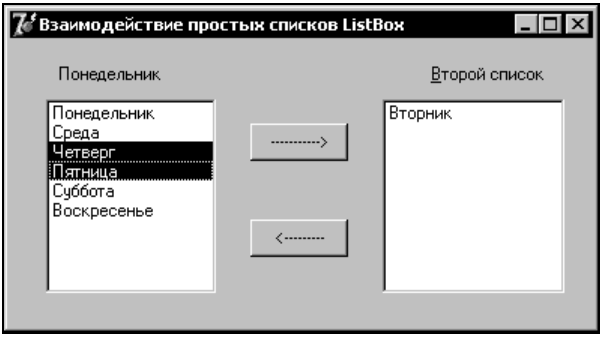


Рис. 3.12. Организация взаимодействия списков

Текст модуля формы приложения приведен в листинге 3.2.

Листинг 3.2. Пример взаимодействия простых списков

```
unit unit5;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TForm5 = class(TForm)
    ListBox1: TListBox;
    ListBox2: TListBox;
    btnRight: TButton;
    btnLeft:  TButton;
    Label1:   TLabel;
    Label2:   TLabel;
    procedure btnRightClick(Sender: TObject);
    procedure btnLeftClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure ListBox2DragOver(Sender, Source: TObject;
              X, Y: Integer; State: TDragState; var Accept: Boolean);
    procedure ListBox2DragDrop(Sender, Source: TObject;
              X, Y: Integer);
    procedure ListBox1DragOver(Sender, Source: TObject;
              X, Y: Integer;
      State: TDragState; var Accept: Boolean);
    procedure ListBox1DragDrop(Sender, Source: TObject;
              X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form5: TForm5;
implementation
{$R *.DFM}

procedure TForm5.FormCreate(Sender: TObject);
begin
  Label1.FocusControl := ListBox1;
  Label2.FocusControl := ListBox2;

  // Отключение сортировки элементов
  ListBox1.Sorted := False;
  ListBox2.Sorted := False;

  // Включение множественного выбора элементов
  ListBox1.MultiSelect := True;
  ListBox2.MultiSelect := True;
```

```
  // Разрешение выбора элементов с помощью клавиатуры
  ListBox1.ExtendedSelect := True;
  ListBox2.ExtendedSelect := True;

  // Заполнение первого списка
  ListBox1.Clear;
  ListBox1.Items.Add('Понедельник');
  ListBox1.Items.Add('Вторник');
  ListBox1.Items.Add('Среда');
  ListBox1.Items.Add('Четверг');
  ListBox1.Items.Add('Пятница');
  ListBox1.Items.Add('Суббота');
  ListBox1.Items.Add('Воскресенье');

  // Очистка второго списка
  ListBox2.Clear;

  // Разрешение автоматического начала операции
  // перемещения элементов мышью
  ListBox1.DragMode := dmAutomatic;
  ListBox2.DragMode := dmAutomatic;
end;

// Перенос элементов во второй список
procedure TForm5.btnRightClick(Sender: TObject);
var i :integer;
begin
  for i := ListBox1.Items.Count — 1 downto 0 do
    if ListBox1.Selected[i] then begin
       ListBox2.Items.Add(ListBox1.Items[i]);
       ListBox1.Items.Delete(i);
    end;
end;

// Перенос элементов в первый список
procedure TForm5.btnLeftClick(Sender: TObject);
var i :integer;
begin
  for i := ListBox2.Items.Count — 1 downto 0 do
    if ListBox2.Selected[i] then begin
       ListBox1.Items.Add(ListBox2.Items[i]);
       ListBox2.Items.Delete(i);
    end;
end;

// Разрешение/запрет перетаскивания мышью элемента во второй список
procedure TForm5.ListBox2DragOver(Sender, Source: TObject;
            X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source = ListBox1 then Accept := True else Accept := False;
end;
```

```
// Перемещение элемента во второй список после отпускания кнопки мыши
procedure TForm5.ListBox2DragDrop(Sender, Source: TObject;
                                  X, Y: Integer);
begin
  with Source as TListBox do begin
    ListBox2.Items.Add(Items[ItemIndex]);
    Items.Delete(ItemIndex);
  end;
end;

// Разрешение/запрет перетаскивания мышью элементов в первый список
procedure TForm5.ListBox1DragOver(Sender, Source: TObject;
             X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source = ListBox2 then Accept := True else Accept := False;
end;

// Перемещение элемента в первый список после отпускания кнопки мыши
procedure TForm5.ListBox1DragDrop(Sender, Source: TObject;
                                  X, Y: Integer);
begin
  with Source as TListBox do begin
    ListBox1.Items.Add(Items[ItemIndex]);
    Items.Delete(ItemIndex);
  end;
end;
end.
```

При нажатии кнопки `btnRight` выбранные в первом списке элементы в цикле добавляются ко второму списку, после чего происходит их удаление из первого списка. При организации цикла перебор элементов списка выполняется с конца, в противном случае произойдет ошибка из-за того, что элемент уже удален, а число повторов цикла не уменьшилось. Для проверки, выбран элемент или нет, используется свойство `Selected` каждого элемента списка.

Перемещение элементов в первый список, выполняемое при нажатии кнопки `btnLeft`, происходит аналогично.

В данном примере перетаскивание элементов мышью имеет особенность, заключающуюся в том, что перемещается элемент, выбранный последним. Это обусловлено тем, что в обработчиках события `DragDrop` при анализе выбора элемента использовано свойство `ItemIndex`, а не `Selected`.

Чтобы перетаскивание элементов мышью выполнялось так же, как и перемещение при нажатии кнопки, в обработчиках события `DragDrop` можно разместить тот же код, что и в обработчиках события `OnClick` соответствующих кнопок. Более предпочтительным является вызов обработчика события `OnClick` кнопки из обработчика события `DragDrop`, например, так:

```
procedure TForm5.ListBox2DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  btnRight.Click;
end;
```

или так:

```
procedure TForm5.ListBox2DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  btnRightClick(Sender);
end;
```

Поскольку выполняемые со списками операции отличаются только названиями компонентов `ListBox1` или `ListBox2`, для события `OnClick` кнопок и событий `DragOver` и `DragDrop` списков можно подготовить общий обработчик (см. пример в *разд. "Однострочные редакторы" ранее в данной главе*).

Работа с кнопками

Кнопки являются элементами управления и служат для выдачи команд на выполнение определенных функциональных действий, поэтому часто их еще называют *командными кнопками*. На поверхности кнопки могут располагаться текст и/или графическое изображение.

Система Delphi предлагает несколько компонентов, представляющих собой различные варианты кнопок:

- стандартная кнопка `Button`;
- кнопка `BitBtn` с рисунком;
- кнопка `SpeedButton` быстрого доступа.

Как элементы управления, эти виды кнопок появились в разное время, но имеют много общего. Различия в облике и функциональных возможностях между разными вариантами кнопок незначительны.

Стандартная кнопка

Стандартная кнопка, или просто *кнопка*, представлена в Delphi компонентом `Button`, который является оконным элементом управления.

Кнопка `Button` может иметь на поверхности надпись (назначение кнопки или описание действий, выполняемых при ее нажатии).

Основным для кнопки является событие `OnClick`, возникающее при ее нажатии. При этом кнопка принимает соответствующий вид, подтверждая происходящее действие визуально. Действия, выполняемые в обработчике события `OnClick`, происходят сразу после отпускания кнопки.

Кнопку можно нажать следующими способами:

- щелчком мыши;
- выбором комбинации клавиш, если она задана в свойстве `Caption`;
- нажатием клавиш <Enter> или <Пробел>;
- нажатием клавиши <Esc>.

На нажатие клавиш <Enter> или <Пробел> реагирует *кнопка по умолчанию*, т. е. находящаяся в фокусе кнопка, заголовок которой выделен пунктирным прямоугольником (рис. 3.13).

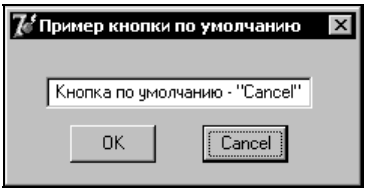


Рис. 3.13. Кнопка по умолчанию (**Cancel**)

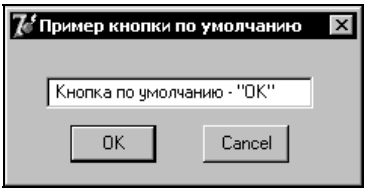


Рис. 3.14. Еще одна кнопка по умолчанию (**OK**)

Если фокус ввода получает некнопочный элемент управления, например, `Edit` или `Memo`, то кнопкой по умолчанию становится та, у которой свойство `Default` типа `Boolean` установлено в значение `True`. В этом случае кнопка по умолчанию выделяется черным прямоугольником (кнопка **OK** в диалоговом окне на рис. 3.14). При размещении в процессе конструирования приложения кнопок в форме (или в другом контейнере, например, `Panel`) это свойство имеет значение `False`, т. е. выбранных кнопок нет. Если свойство `Default` программно установить в значение `True` для двух и более кнопок, это не приведет к ошибке, но кнопкой по умолчанию будет являться первая кнопка по порядку обхода при табуляции.

Событие `OnClick` может генерироваться для кнопки и в случае нажатия клавиши <Esc>, что обычно реализуется для кнопок, связанных с отменой какого-либо действия, например, кнопка **Cancel** в диалоговом окне. Чтобы кнопка реагировала на нажатие клавиши <Esc>, необходимо ее свойство `Cancel` типа `Boolean` установить в значение `True`. При установке значения `True` для свойств `Cancel` двух и более кнопок кнопкой отмены считается первая кнопка по порядку обхода при табуляции. По умолчанию значение свойства `Cancel` равно `False`, и никакая кнопка не реагирует на нажатие клавиши <Esc>.

Замечание

Если в фокусе ввода находится некнопочный элемент управления, например редактор `Memo`, то он первый получает сообщение о нажатии клавиши и, соответственно, первым может реагировать на нажатие таких клавиш, как <Enter> или <Esc>, обрабатывая их по-своему.

При применении кнопки для *закрытия окна* можно использовать ее свойство `ModalResult` типа `TModalResult`. Оно определяет, какое значение будет содержать одноименное свойство `ModalResult` формы, когда окно закрывается при нажатии соответствующей кнопки. Обычно свойство `ModalResult` применяется для закрытия модальных окон, и его возможными значениями являются целые числа, некоторые из них объявлены как именованные константы:

- `mrNone` (число `0`);
- `mrOk`, имеет значение `idOK` (число `1`);

- `mrCancel`, имеет значение `idCancel` (число `2`);
- `mrAbort`, имеет значение `idAbort` (число `3`);
- `mrRetry`, имеет значение `idRetry` (число `4`);
- `mrIgnore`, имеет значение `idIgnore` (число `5`);
- `mrYes`, имеет значение `idYes` (число `6`);
- `mrNo`, имеет значение `idNo` (число `7`);
- `mrAll`, имеет значение `mrNo + 1`;
- `mrNoToAll`, имеет значение `mrAll + 1`;
- `mrYesToAll` — `mrNoToAll + 1`.

Если свойство `ModalResult` кнопки установлено в ненулевое значение, отличное от `mrNone` (по умолчанию), то при нажатии кнопки модальная форма закрывается автоматически, поэтому нет необходимости вызывать метод `Close` в обработчике события `OnClick` этой кнопки. Например:

```
procedure TForm2.FormCreate(Sender: TObject);
begin
  Button2.ModalResult := mrOK;
  Button3.ModalResult := 123;
end;

procedure TForm2.Button1Click(Sender: TObject);
begin
  // Эта инструкция не нужна
  Button1.ModalResult := mrNone;
  Form2.Close;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
  // Для закрытия формы код не нужен
end;

procedure TForm2.Button3Click(Sender: TObject);
begin
  // Для закрытия формы код не нужен
end;
```

При нажатии любой из кнопок `Button1`, `Button2` или `Button3` модальная форма `Form2` закрывается, причем для этого не требуются обработчики события нажатия кнопок `Button2` или `Button3`.

Обычно требуемые значения свойства `ModalResult` для кнопок устанавливаются при проектировании формы с помощью Инспектора объектов. Однако их можно задать и при выполнении приложения, что может понадобиться, если необходимо запретить закрытие диалоговой формы, например, при наличии несохраненных редактируемых данных:

```
procedure TForm2.CheckMemo(Sender: TObject);
begin
  if Memo1.Modified
     then Button3.ModalResult := mrNone
     else Button3.ModalResult := mrOK
end;
```

Более подробно использование форм рассматривается в *главе 4*.

Кнопка с рисунком

Кнопка с рисунком в Delphi представлена компонентом `BitBtn`, класс которой `TBitBtn` порожден непосредственно от класса `TButton` стандартной кнопки `Button`. Кнопка с рисунком отличается от стандартной кнопки тем, что помимо заголовка на ней можно отобразить растровое изображение. Видом и размещением изображения на поверхности кнопки `BitBtn` можно управлять с помощью свойств.

Свойство `Glyph` типа `TBitmap` определяет растровый рисунок кнопки. По умолчанию свойство `Glyph` имеет значение `None`, т. е. кнопка не содержит рисунок. Рисунок может содержать до трех отдельных изображений (глифов). Какое именно изображение выводится на кнопке, зависит от ее текущего состояния:

- первое изображение отображается, если кнопка не нажата (по умолчанию);
- второе изображение отображается, если кнопка неактивна и не может быть выбрана;
- третье изображение отображается, когда кнопка нажата (выполнен щелчок).

При использовании нескольких изображений они должны быть подготовлены и сохранены в одном файле растрового формата BMP. Подготовить рисунок для кнопки можно в графическом редакторе. На рис. 3.15 показана подготовка рисунка из трех изображений с помощью редактора Image Editor, входящего в состав Delphi. Все отдельные изображения в рисунке должны располагаться без промежутков в горизонтальной строке и иметь одинаковую высоту и ширину (как правило, 16×16 пикселов). По умолчанию левый нижний пиксел каждого рисунка определяет фоновый цвет рисунка. Обычно ему задают цвет поверхности кнопки (значение `clBtnFace`), при этом все пикселы изображения, имеющие тот же цвет, будут не видны, т. е. являются прозрачными. По этой причине фоновый цвет также называют *прозрачным*. Если установить, например, желтый цвет фонового пиксела, то фон изображения тоже станет желтым.

Изменить *режим отображения* картинки в случае, когда фоновый цвет задается левым нижним пикселом рисунка, можно, установив значения его взаимосвязанных свойств `TransparentColor` и `TransparentMode`. Для восстановления режима отображения по умолчанию нужно установить свойство `TransparentMode` в значение `tmAuto`.

Используя рисунок с несколькими различными изображениями, можно при нажатии кнопки воспроизводить на ее поверхности простейшую *анимацию*. Для этого первое и третье изображения должны различаться не только цветом, но и видом и расположением фигур. Отметим, что для реализации более сложной анимации можно использовать компонент `Animate`, расположенный непосредственно на кнопке.

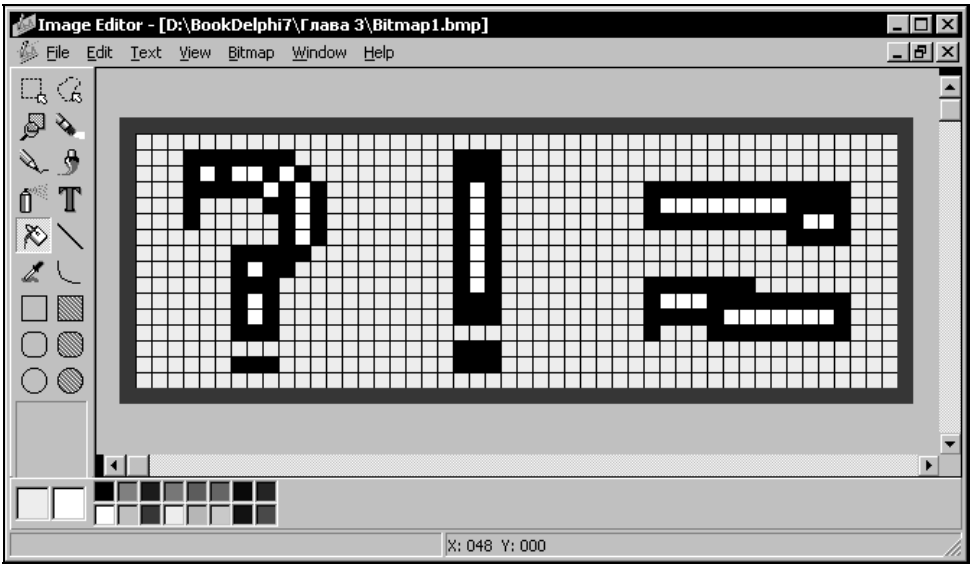


Рис. 3.15. Подготовка рисунка для кнопки

Количество изображений указывается в свойстве `NumGlyph` типа `TNumGlyphs`. По умолчанию свойство `NumGlyph` имеет значение `1`, и на кнопке всегда отображается первое изображение.

Delphi предлагает для кнопки `BitBtn` несколько предопределенных *видов* (рис. 3.16), выбираемых с помощью свойства `Kind` типа `TBitBtnKind`. При выборе какого-либо вида для кнопки на ней отображается соответствующий глиф. Для задания вида кнопки могут использоваться следующие константы:

- `bkCustom` — на кнопке имеется выбранное изображение; первоначально изображение отсутствует, и его нужно загружать дополнительно;
- `bkOK` — на кнопке имеются глиф "зеленая галочка" и текст **OK**; свойство `Default` кнопки установлено в значение `True`, а свойство `ModalResult` — в значение `mrOK`;
- `bkCancel` — на кнопке имеются глиф "красный знак ×" и текст **Cancel**; свойство `Cancel` кнопки установлено в значение `True`, а свойство `ModalResult` — в значение `mrCancel`;
- `bkYes` — на кнопке имеются глиф "зеленая галочка" и текст **Yes**; свойство `Default` кнопки установлено в значение `True`, а свойство `ModalResult` — в значение `mrYes`;

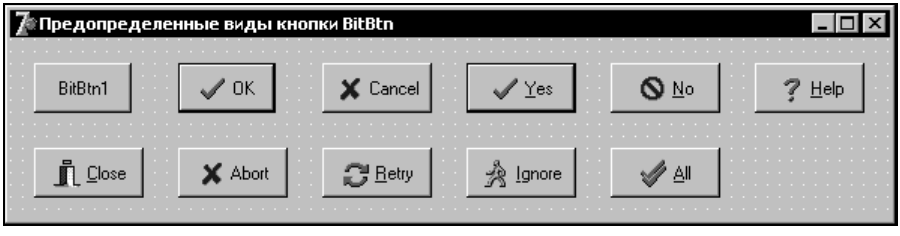


Рис. 3.16. Предопределенные виды кнопок `BitBtn`

- `bkNo` — на кнопке имеются глиф "красная перечеркнутая окружность" и текст **No**; свойства `Cancel` кнопки установлено в значение `True`, а свойство `ModalResult` — в значение `mrNo`;
- `bkHelp` — на кнопке имеются глиф "сине-зеленый вопросительный знак" и текст **Help**;
- `bkClose` — на кнопке имеются глиф "дверь с обозначением выхода" и текст **Close**; при нажатии кнопки форма автоматически закрывается;
- `bkAbort` — на кнопке имеются глиф "красный знак ×" и текст **Abort**;
- `bkRetry` — на кнопке имеются глиф "зеленая стрелка повтора операции" и текст **Retry**;
- `bkIgnore` — на кнопке имеются глиф "игнорирование" и текст **Ignore**;
- `bkAll` — на кнопке имеются глиф "двойная зеленая галочка" и текст **Yes to All**.

По умолчанию свойство `Kind` имеет значение `bkCustom`, и пользователь может сам выбирать изображение, управляя свойством. Не рекомендуется изменять свойство `Glyph` для предопределенных кнопок (например, для кнопки **Close)**, т. к. в этом случае кнопка не будет выполнять закрепленные за ней действия (в данном случае закрытие окна).

Расположением изображения на поверхности кнопки относительно текста (рис. 3.17) управляет свойство `Layout` типа `TButtonLayout`, принимающее следующие значения:

- `blGlyphLeft` (изображение слева от текста) — по умолчанию;
- `blGlyphRight` (изображение справа от текста);
- `blGlyphTop` (изображение над текстом);
- `blGlyphBottom` (изображение под текстом).

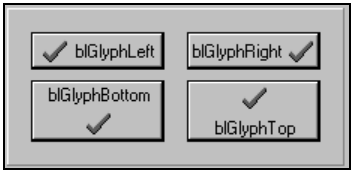


Рис. 3.17. Варианты размещения изображения на поверхности кнопки

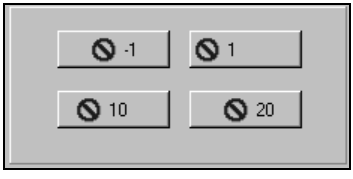


Рис. 3.18. Варианты выравнивания изображения относительно сторон кнопки

С помощью свойства `Margin` типа `Integer` можно управлять *выравниванием* глифа и текста относительно сторон кнопки. Это свойство задает расстояние в пикселах между стороной кнопки и изображением и по умолчанию имеет значение `-1`, что означает расположение глифа и текста по центру кнопки. Сторона, относительно которой производится выравнивание, определяется свойством `Layout`. Например, если значение `Layout` равно `blGlyphLeft`, то выравнивание выполняется по левой стороне кнопки. На рис. 3.18 показаны варианты выравнивания изображения и текста, соответствующие разным значениям свойства `Margin`.

Свойство `Spacing` типа `Integer` определяет *размер (в пикселах) промежутка*, отделяющего глиф от текста. По умолчанию значение этого свойства равно `4` пикселам. Если

значение этого свойства равно -1, то имеет место центрирование текста между краем глифа и дальней от него стороной кнопки. На рис. 3.19 показано использование различных значений свойства Spacing, отображенных в виде текста на кнопках.

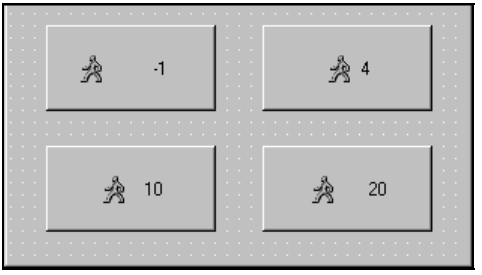


Рис. 3.19. Управление расстоянием между глифом и текстом

Кнопка быстрого доступа

Кнопка быстрого доступа представлена в Delphi компонентом SpeedButton, который по своему виду и функциональным возможностям в общем похож на кнопку с рисунком BitBtn. Однако, в отличие от кнопки с рисунком, кнопка SpeedButton происходит от класса TGraphicControl и является *неоконным элементом управления*. Поэтому кнопка быстрого доступа не может получать фокус ввода, но зато требует для своего функционирования меньше ресурсов, чем другие виды кнопок. Наиболее часто кнопки быстрого доступа, или быстрые кнопки, применяются для создания панелей инструментов.

В отличие от других кнопок, кнопка SpeedButton может использоваться как переключатель. Поэтому, помимо обычного и нажатого состояний, она имеет третье состояние — утопленное, или выбранное (включенное). *Включена* быстрая кнопка или нет, определяет свойство Down типа Boolean. Если свойство имеет значение True, то кнопка выбрана, если False — не выбрана.

Использование быстрых кнопок в качестве переключателей имеет определенные особенности. Все быстрые кнопки объединяются в группы, и каждая такая кнопка должна принадлежать к одной из групп. *Принадлежность кнопки к группе* определяет свойство GroupIndex типа Integer. По умолчанию свойство имеет нулевое значение, и быстрая кнопка не относится к группе. Определяя одинаковое значение для свойств GroupIndex различных кнопок, их можно сгруппировать. При этом все кнопки группы будут работать согласованно — если одна из кнопок выбрана и находится в фиксированном нижнем положении, то выбор других автоматически отменяется. Группирование кнопок снимает необходимость организовывать взаимодействие элементов управления вручную. На рис. 3.20 показана панель инструментов **Форматирование** текстового процессора Microsoft Word.

Рис. 3.20. Панель инструментов **Форматирование** текстового процессора Microsoft Word

На этой панели зависимыми кнопками, входящими в одну группу, являются кнопки управления выравниванием текста (рис. 3.21).

Рис. 3.21. Взаимозависимые (сгруппированные) кнопки быстрого доступа

Для случая, когда быстрая кнопка может быть выбрана, свойство `AllowAllUp` типа `Boolean` определяет, можно ли повторным щелчком вернуть эту кнопку в невыбранное состояние. Если свойство `AllowAllUp` имеет значение `True`, то такое переключение возможно, в противном случае кнопка выключается только при выборе другой кнопки в составе группы. По умолчанию свойство `AllowAllUp` имеет значение `False`.

Если кнопка не входит в группу (`GroupIndex = 0`), то она не может работать как переключатель и находиться в выбранном состоянии. Поэтому в случае, когда необходимо, чтобы быстрая кнопка могла находиться во включенном состоянии и работала независимо от других кнопок, создается группа из одной кнопки. С этой целью значению свойства `GroupIndex` такой быстрой кнопки присваивается уникальный номер, а свойство `AllowAllUp` устанавливается в значение `True`.

На панели инструментов **Форматирование** текстового процессора Microsoft Word независимыми являются, например, кнопки управления начертанием шрифта (рис. 3.22).

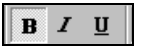


Рис. 3.22. Независимые кнопки быстрого доступа

В связи с тем, что быстрая кнопка по сравнению с кнопкой `BitBtn` имеет еще одно состояние, рисунок на ее поверхности может состоять не из трех, а из четырех отдельных изображений. Четвертое изображение на ее поверхности появляется в случае, когда кнопка находится в утопленном (выбранном) состоянии (свойство `Down` имеет значение `True`). Поэтому для кнопки `SpeedButton` максимальное значение свойства `NumGlyph` равно `4`.

Использование переключателей и флажков

Переключатель (зависимый переключатель) позволяет выбрать единственное значение из определенного множества значений, представленного группой переключателей. Он может находиться в выбранном или невыбранном состоянии. Одновременно можно выбрать только один переключатель в группе; одиночный переключатель никогда не используется.

Флажок (независимый переключатель) отличается от переключателя тем, что в группе флажков одновременно можно установить флажки в любой комбинации (в том числе могут быть установлены или сброшены все флажки и т. д.). Флажок может находиться в установленном или сброшенном состоянии. Одиночный флажок часто используется, например, для включения/выключения какого-либо режима.

Анализ состояния переключателя или флажка позволяет программисту выполнять соответствующие операции.

Система Delphi предоставляет для работы с переключателями компоненты `CheckBox`, `RadioButton` и `RadioGroup`. Классы компонентов `CheckBox` и `RadioButton`, как и кнопка `Button`, происходят от класса `TButtonControl`. Поэтому иногда эти переключатели называют кнопками с фиксацией: `CheckBox` — с независимой фиксацией, а `RadioButton` — с зависимой.

Флажок

Флажок представлен компонентом `CheckBox`. Флажок действует независимо от других флажков, несмотря на то, что по функциональному назначению их часто объединяют в группы с общим названием, например, **Show** и **Formatting marks** на рис. 3.23.

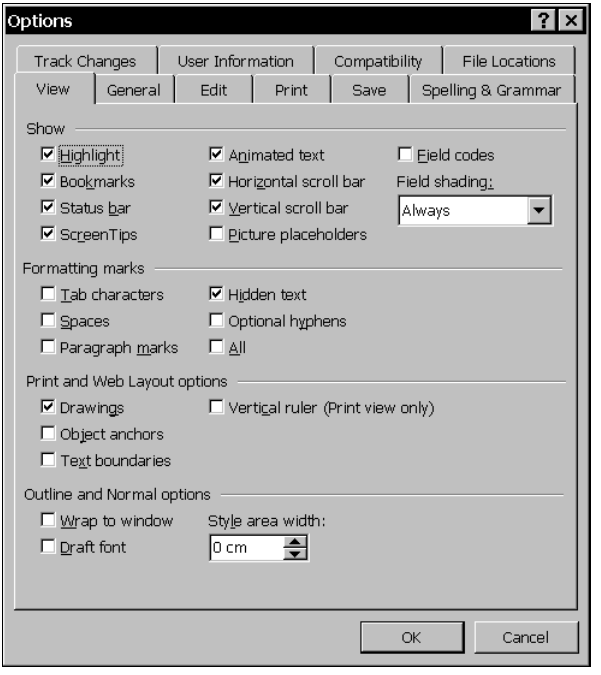


Рис. 3.23. Различные группы флажков

Флажок выглядит как прямоугольник с текстовым заголовком. Если в нем есть галочка, то обозначенная этим флажком опция включена (в этом случае также говорят, что флажок *отмечен*). Если прямоугольник пуст, то флажок снят, или сброшен. Действия с одним флажком не отражаются на состоянии других флажков, если это не было специально предусмотрено, что на практике применяется редко.

Для *определения состояния* флажка используется свойство `Checked` типа `Boolean`. По умолчанию оно имеет значение `False`, и флажок снят.

Пользователь может переключать состояние флажка щелчком мыши. Если флажок снят (не включен), то после щелчка он будет установлен (включен), и наоборот. При

этом соответственно изменяется значение свойства `Checked`. Флажок можно переключить и с помощью клавиши <Пробел>, когда компонент `CheckBox` находится в фокусе ввода, а вокруг его заголовка отображен черный пунктирный прямоугольник.

В примере

```
if CheckBox1.Checked then
   MessageDlg('Время истекло!', mtError, [mbOK], 0);
```

сообщение `Время истекло!` выдается при включенном состоянии флажка `CheckBox1`, который регулирует выдачу сообщения об истечении лимита времени.

Флажком можно управлять программно, устанавливая свойство `Checked` в требуемые значения. Например:

```
CheckBox2.Checked := True;
CheckBox3.Checked := False;
```

Сделать флажок недоступным для изменения (заблокировать) можно установив свойство `Enabled` в значение `False`:

```
CheckBox1.Enabled := False;
```

После перехода флажка в заблокированный режим он сохраняет то состояние, в котором находился до выполнения блокировки. То есть неактивный флажок может находиться как в установленном, так и в снятом состояниях.

Кроме двух состояний (установлен/снят) флажок может иметь и третье состояние — запрещенное, или недоступное. Наличием или отсутствием этого состояния управляет свойство `AllowGrayed` типа `Boolean`. Если оно имеет значение `True`, то при щелчке мышью происходит циклический переход между тремя состояниями флажка: установлен, снят и недоступен. В недоступном состоянии флажок выделен серым цветом, а в прямоугольнике находится знак галочки.

Замечание

Галочка, отображаемая флажком в недоступном состоянии, способна ввести в заблуждение, т. к. подобное состояние можно интерпретировать как включенное.

Свойство `Checked` имеет значение `True` только для выбранного режима флажка.

Для *анализа* и *установки* одного из трех состояний флажка (рис. 3.24) служит свойство `State` типа `TCheckBoxState`. Оно может принимать следующие значения:

- `cbUnchecked` (флажок не включен);
- `cbChecked` (флажок включен);
- `cbGrayed` (флажок недоступен).

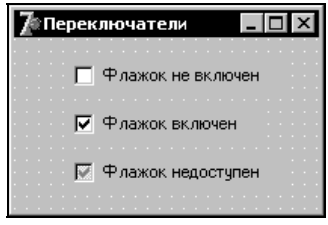


Рис. 3.24. Состояния флажка (компонента `CheckBox`)

При изменении состояния флажка возникает событие `OnClick`, независимо от того, в какое состояние он переходит. В обработчике события `OnClick` обычно располагаются инструкции, проверяющие состояние флажка и выполняющие требуемые действия.

Приведем в качестве примера процедуру, производящую обработку события выбора состояния флажка.

```
procedure TForm1.CheckBox3Click(Sender: TObject);
begin
  case CheckBox3.State of
     cbUnchecked: CheckBox3.Caption := 'Флажок не включен';
     cbChecked:   CheckBox3.Caption := 'Флажок включен';
     cbGrayed:    CheckBox3.Caption := 'Флажок недоступен';
  end;
end;
```

Флажок `CheckBox3` при его переключении отображает в заголовке свое состояние.

Переключатель

Переключатель представлен компонентом `RadioButton`. Соответствующие элементы управления отображаются в виде кружка с текстовым заголовком (при выбранном состоянии в этом кружке появляется черная точка).

Переключатели обычно располагаются по группам, визуально выделенным в форме. Выбор переключателя является взаимоисключающим, т. е. при выборе одного переключателя другие становятся невыбранными. Delphi поддерживает автоматическое группирование переключателей. Каждый переключатель, помещенный в контейнер, включается в находящуюся на нем группу (рис. 3.25). Контейнерами обычно служат такие компоненты, как форма `Form`, панель `Panel`, группа `GroupBox` и область прокрутки `ScrollBox`.

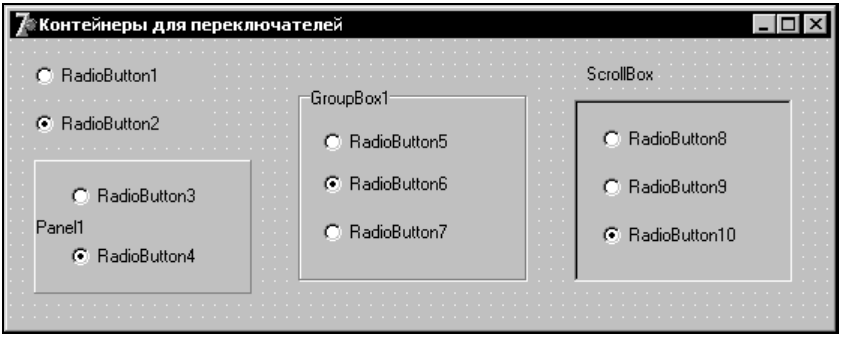


Рис. 3.25. Виды контейнеров для переключателей

При работе с группой один из переключателей рекомендуется делать выбранным по умолчанию, что можно выполнить при проектировании формы или в процессе выполнения приложения. Например, для приведенной на рис. 3.25 формы это можно выполнить так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Все переключатели расположены в разных группах
  RadioButton2.Checked := True;
  RadioButton4.Checked := True;
  RadioButton6.Checked := True;
  RadioButton10.Checked := True;
end;
```

Когда в группе выбран один из переключателей, то, в отличие от флажка, его состояние нельзя изменить повторным щелчком. Отмена выбора переключателя происходит только при выборе другого переключателя из этой же группы.

Замечание

Для компонента `RadioButton` событие `OnClick` генерируется только при выборе переключателя. Повторный щелчок на переключателе не приводит к возникновению события `OnClick`.

Кроме уже упомянутых элементов-контейнеров, объединяющих переключатели в группу, в Delphi есть специализированный компонент `RadioGroup` (рис. 3.26), представляющий собой группу переключателей `RadioButton`. Такая группа переключателей создана для упорядочения переключателей и упрощения организации их взаимодействия по сравнению с добавлением их вручную к обычной группе.

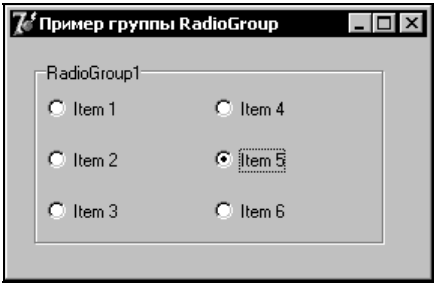


Рис. 3.26. Группа переключателей `RadioGroup`

Группа переключателей `RadioGroup` может также содержать другие элементы управления, например, флажок `CheckBox` или однострочный редактор `Edit`.

Управление числом и *названиями* переключателей производится с помощью свойства `Items` типа `TStrings`, которое позволяет получить доступ к отдельным переключателям в группе. Это свойство содержит строки, отображаемые как заголовки переключателей. Отсчет строк в массиве начинается с нуля: `Items[0]`, `Items[1]` и т. д. Для манипуляции строками (заголовками) можно использовать такие методы, как `Add` и `Delete`.

Доступ к отдельному переключателю можно получить через свойство `ItemIndex` типа `Integer`, содержащее позицию (номер) переключателя, выбранного в группе в текущий момент. Это свойство используется для выбора отдельного переключателя или для определения, какой из переключателей является выбранным. По умолчанию свойство `ItemIndex` имеет значение `-1` (не выбран ни один переключатель).

Свойство `Columns` типа `Integer` задает *число столбцов*, на которое разбиваются переключатели при расположении в группе (по умолчанию `1`). Это свойство действует только на переключатели, принадлежащие массиву `Items` группы, и не действует на другие элементы управления, например, на однострочный редактор `Edit` или надпись `Label`, размещенные в группе `RadioGroup`.

Пример работы с группой переключателей:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  RadioGroup1.Items.Clear;
  RadioGroup1.Items.Add('Item 1');
  RadioGroup1.Items.Add('Item 2');
  RadioGroup1.Items.Add('Item 3');
  RadioGroup1.Items.Add('Item 4');
  RadioGroup1.Items.Add('Item 5');
  RadioGroup1.Items.Add('Item 6');
  RadioGroup1.Columns := 2;
  RadioGroup1.ItemIndex := 4;
end;
```

При создании формы в группу `RadioGroup1` включаются 6 переключателей, расположенных в два столбца. Переключатель с заголовком `Item 5` становится выбранным.

Объединение элементов управления

При разработке приложения часто возникает задача объединения, или группирования, различных элементов управления. Группирование может понадобиться, например, при работе с переключателями в форме или при создании панели инструментов.

Объединение элементов выполняется с помощью специальных компонентов — контейнеров. *Контейнер* представляет собой визуальный компонент, на поверхности которого можно размещать другие компоненты; контейнер объединяет эти компоненты в группу и становится их владельцем. Владелец также отвечает за прорисовку своих дочерних элементов. Дочерний элемент может *ссылаться на владельца* с помощью свойства `Parent`.

В предыдущем разделе мы рассмотрели специализированный компонент — контейнер `RadioGroup`, используемый для организации группы переключателей. Для различных объектов система Delphi предлагает также набор *универсальных контейнеров*, в который входят такие компоненты, как:

- группа `GroupBox`;
- панель `Panel`;
- область прокрутки `ScrollBox`;
- фрейм (рамка) `Frame`.

Отметим, что форма также является контейнером, с которого обычно и начинается конструирование интерфейсной части приложения. Форма — владелец всех расположенных на ней компонентов.

Группа

Группа используется в основном для визуального выделения функционально связанных управляющих элементов. Для работы с группой Delphi предоставляет компонент `GroupBox`, задающий прямоугольную рамку с заголовком (свойство `Caption`) в левом верхнем углу и объединяющий содержащиеся в нем элементы управления. Например, на рис. 3.23 группа с заголовком **Show** используется для отображения функциональной зависимости между полем **Field shading** и девятью флажками, устанавливающими различные параметры. На рис. 3.25 группа используется для объединения переключателей `RadioButton`.

Панель

Панель представляет собой контейнер, в котором можно размещать другие элементы управления. Панели применяются в качестве визуальных средств группирования, а также для создания панелей инструментов и строк состояния. Для работы с панелями в Delphi предназначен компонент `Panel`.

Панель имеет край с двойной фаской: внутренней и внешней. Внутренняя фаска обрамляет панель, а внешняя отображается вокруг внутренней.

Ширина каждой фаски в пикселах задается свойством `BewelWidth` типа `TBewelWidth`. Значение типа `TBevelWidth` представляет собой целое число (`TBevelWidth = 1..MaxInt`). По умолчанию ширина фаски равна `1`.

Свойства `BevelInner` и `BevelOuter` типа `TPanelBevel` определяют *вид внутренней* и *внешней фасок* соответственно. Каждое из свойств может принимать следующие значения:

- `bvNone` (нет фаски);
- `bvLowered` (фаска утоплена);
- `bvRaised` (фаска приподнята);
- `bvSpace` (действие неизвестно).

По умолчанию свойство `BevelInner` имеет значение `bvNone`, а свойство `BevelOuter` — значение `bvRaised`.

Между фасками может быть промежуток, *ширина* которого в пикселах определяется свойством `BorderWidth` типа `TBorderWidth`. По умолчанию ширина промежутка равна нулю (промежутка нет).

На рис. 3.27 показаны различные виды панелей в зависимости от установки свойств `BevelInner`, `BevelOuter`, `BorderWidth` и `BevelWidth`, значения которых приведены в табл. 3.3.

Таблица 3.3. *Значения свойств панелей*

Панель	BevelInner	BevelOuter	BevelWidth	BorderWidth
Panel1	bvNone	bvRaised	1	00
Panel2	bvNone	bvRaised	5	00

Таблица 3.3 *(окончание)*

Панель	BevelInner	BevelOuter	BevelWidth	BorderWidth
Panel3	bvLowered	bvRaised	1	00
Panel4	bvNone	bvLowered	1	00
Panel5	bvNone	bvLowered	5	00
Panel6	bvRaised	bvLowered	1	00
Panel7	bvRaised	bvRaised	5	05
Panel8	bvLowered	bvLowered	5	05
Panel9	bvLowered	bvRaised	1	10

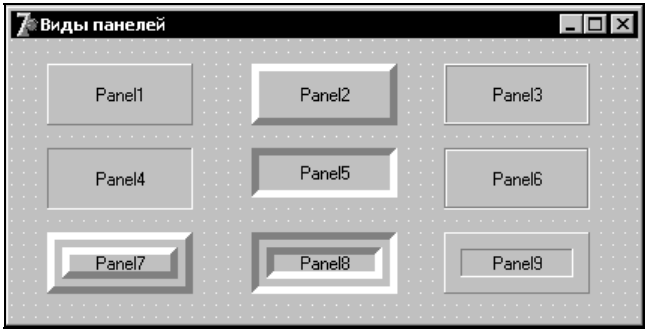


Рис. 3.27. Виды панелей с различными типами фасок

Управление расположением заголовка панели осуществляется с помощью свойства `Alignment` типа `TAlignment`, которое может принимать следующие значения:

- `taLeftJustify` (выравнивание по левому краю);
- `taCenter` (выравнивание по центру) — по умолчанию;
- `taRightJustify` (выравнивание по правому краю).

Если заголовок панели не нужен, то значением свойства `Caption` должна быть пустая строка.

Область прокрутки

Область прокрутки представляет собой окно с возможностью прокрутки информации. Внутри нее размещаются другие элементы управления. В Delphi область прокрутки представлена компонентом `ScrollBox`.

Компонент `ScrollBox` является элементом управления, поверхность которого может быть больше той части, которую в данный момент видит пользователь. Если какой-либо элемент, содержащийся в компоненте `ScrollBox`, виден не полностью, то автоматически могут появляться полосы прокрутки: горизонтальная (рис. 3.28, слева), вертикальная или обе одновременно. Если размеры области увеличиваются, полосы прокрутки могут автоматически исчезать (рис. 3.28, справа), если в них нет необходимости.

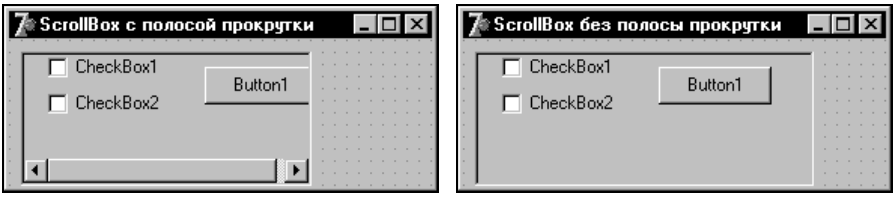


Рис. 3.28. Компоненты `ScrollBox`

Компонент `ScrollBox` удобно использовать, например, в случае, когда форма содержит панель инструментов и строку состояния. Если не все элементы управления полностью видны в отображаемой области окна, то на нем могут присутствовать полосы прокрутки. Однако при использовании горизонтальной полосы прокрутки панель инструментов или строка состояния не будут видны в форме (рис. 3.29, слева).

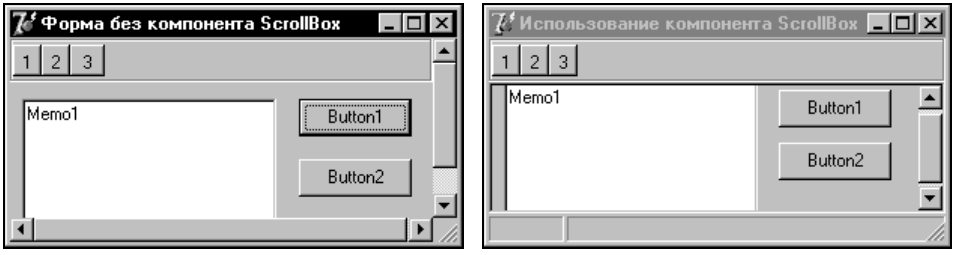


Рис. 3.29. Область с полосами прокрутки

Это не произойдет, если при проектировании расположить в форме компонент `ScrollBox` и установить его свойство `Align` в значение `alClient`. Тогда область прокрутки займет все место формы, не занятое панелью инструментов и строкой состояния, после чего на ней разместятся другие элементы управления. Теперь в случае появления полос прокрутки они будут принадлежать компоненту `ScrollBox`, обеспечивая доступ ко всем элементам управления. В то же время панель инструментов и строка состояния, расположенные выше и ниже области прокрутки, будут видны и доступны для выполнения операций (рис. 3.29, справа).

Свойство `AutoScroll` типа `Boolean` определяет, будут ли при необходимости *автоматически появляться* полосы прокрутки. По умолчанию свойство имеет значение `True`, и область сама управляет своими полосами прокрутки. Если свойство `AutoScroll` установлено в значение `False`, то программист должен отображать полосы прокрутки самостоятельно, управляя свойствами `HorzScrollBar` и `VertScrollBar` типа `TControlScrollBar`. Работа с этими полосами прокрутки аналогична работе с компонентами `ScrollBar` и `TrackBar`.

Область прокрутки может иметь рамку. *Наличие рамки* определяет свойство `BorderStyle` типа `TBorderStyle`, принимающее два возможных значения:

- `bsNone` (нет рамки);
- `bsSingle` (есть рамка) — по умолчанию.

Для *программного управления* областью прокрутки служит метод `ScrollInView`. Процедура `ScrollInView(AControl: TControl)` автоматически изменяет позиции полос прокрутки так, чтобы элемент интерфейса, заданный параметром `AControl`, был виден в отображаемой области. Например, при выполнении процедуры

```
procedure TForm1.Button1Click (Sender: TObject);
begin
  ScrollBox1.ScrollInView(Button3);
end;
```

кнопка `Button3` становится видимой в области компонента `ScrollBox1`.

Фрейм

Фрейм (рамка, frame) представляет собой контейнер для других компонентов, который на этапе разработки создается так же, как форма, но, в отличие от нее, может быть размещен в другом контейнере, например, в форме или панели. Для работы с фреймами в Delphi служит компонент `Frame`.

Работа с фреймом состоит из двух этапов:

- создание и конструирование фрейма;
- размещение созданного фрейма в нужном месте формы или панели.

Создание фрейма выполняется командой **File | New | Frame** (Файл | Новый | Фрейм), в результате которой в состав проекта включается новый фрейм. Фрейм имеет много общего с формой. Так, для каждого фрейма создаются файлы формы и модуля, а вид фрейма на этапе разработки не отличается от вида формы. Конструирование фрейма, как и конструирование формы, заключается в размещении в нем нужных компонентов и в кодировании обработчиков событий. На рис. 3.30 показан фрейм, в котором после его добавления к проекту размещены группа переключателей и кнопка.

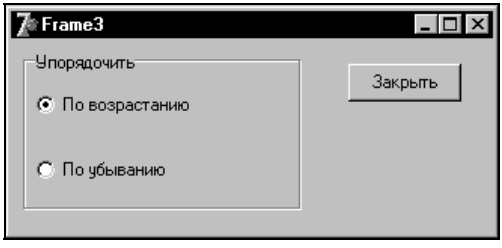


Рис. 3.30. Вид фрейма при разработке

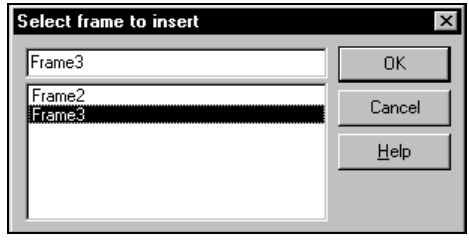


Рис. 3.31. Выбор вставляемого фрейма

Для размещения фрейма в форме следует выбрать в Палитре компонентов компонент `Frame` и поместить его в нужное место формы. При отпускании компонента появляется диалоговое окно **Select frame to insert** (Выберите фрейм для вставки), в котором выбирается имя фрейма (рис. 3.31). После нажатия кнопки **OK** выбранный фрейм вставляется в указанное место. При этом несколько изменяется вид фрейма: у него пропадают

рамка и заголовок, присущие форме. Однако размещенные в нем компоненты (т. е. содержимое фрейма) выглядят так же, как при разработке.

Замечание

Если в проекте нет ни одного фрейма, то при попытке размещения компонента `Frame` в форме выдается сообщение об ошибке.

После размещения фрейма в форме в ее раздел `uses` автоматически включается ссылка на модуль фрейма, а в описание класса формы добавляются ссылки на обработчики событий фрейма и его компонентов.

В случае частого использования какого-либо фрейма его можно разместить в Палитре компонентов. Для этого после создания фрейма следует выделить его и выполнить команду **Create Component Template** (Создать шаблон компонента) меню **Component** (Компонент), что вызовет появление диалогового окна **Component Template Information** (Информация о шаблоне компонента) (рис. 3.32).

После нажатия кнопки **OK** фрейм размещается на странице, выбранной в списке **Palette page** (Страница Палитры), под именем, заданным в поле **Component name** (Имя компонента), и со значком, отображаемым в области **Palette Icon** (Значок палитры). Чтобы сменить значок, нажмите кнопку **Change** (Изменить) и в открывшемся диалоговом окне выберите графический файл с расширением bmp. После размещения в Палитре компонентов с фреймом можно работать, как с любым другим стандартным компонентом, например, панелью или группой.

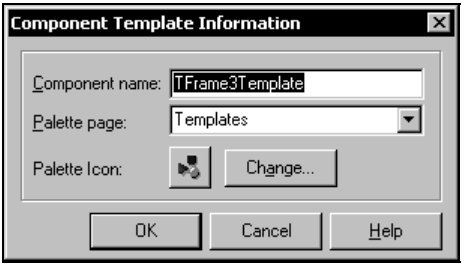


Рис. 3.32. Информация о шаблоне компонента

Можно добавить фрейм и в Хранилище объектов, вызвав командой **Add To Repository** (Добавить к Хранилищу объектов) контекстного меню фрейма одноименное диалоговое окно (рис. 3.33).

Перед добавлением фрейма необходимо указать данные в следующих полях:

- список **Forms** (имя добавляемого фрейма);
- поле **Title** (название фрейма в Хранилище объектов);
- поле **Description** (описание фрейма);
- список **Page** (страница, на которой должен быть размещен фрейм);
- поле **Author** (информация о разработчике фрейма).

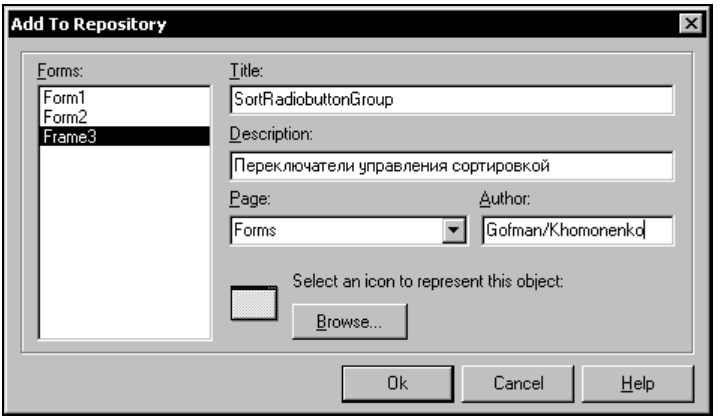
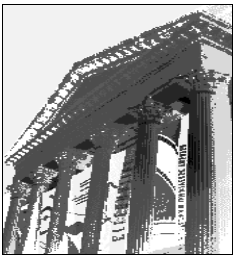


Рис. 3.33. Размещение фрейма в Хранилище объектов

Чтобы сменить значок фрейма, назначаемый по умолчанию, нажмите кнопку **Browse** (Просмотреть) и выберите в открывшемся диалоговом окне файл с расширением ico. После нажатия кнопки **OK** фрейм добавляется в Хранилище объектов.

Аналогичным образом в Хранилище объектов добавляется форма.

ГЛАВА 4



Форма — главный компонент приложения

Форма — это важнейший визуальный компонент. Формы представляют собой видимые окна Windows и являются основной частью практически любого приложения. Термины "форма" и "окно" — синонимы, т. е. обозначают одно и то же.

Для работы с формой предназначен компонент `Form` класса `TForm`. С создания формы начинается конструирование приложения. В форме размещаются визуальные компоненты, образующие интерфейсную часть приложения, и системные (невизуальные) компоненты. Таким образом, в системе Delphi форма является компонентом, который служит *контейнером* для всех других компонентов. В принципе можно создать и безоконное приложение, однако большинство приложений все же имеет видимое на экране окно, содержащее интерфейсную часть приложения.

Приложение может иметь несколько форм, одна из которых считается *главной* и при запуске программы отображается первой. При закрытии главного окна (формы) приложения прекращается работа всего приложения, при этом также закрываются все другие окна приложения. В начале работы над новым проектом Delphi по умолчанию делает главной первую форму (с первоначальным названием `Form1`). В файле проекта (dpr) эта форма создается первой, например:

```
Application.Initialize;
Application.CreateForm(TForm1, Form1);
Application.CreateForm(TForm2, Form2);
Application.Run;
```

Программно можно сделать главной любую форму приложения, первым вызвав метод `CreateForm` создания этой формы. Например, задание формы `Form2` в качестве главной реализуется так:

```
Application.Initialize;
Application.CreateForm(TForm2, Form2);
Application.CreateForm(TForm1, Form1);
Application.Run;
```

При конструировании приложения более удобно указать главную форму в окне параметров проекта, открываемом командой **Project | Options** (Проект | Параметры). Главная

форма выбирается в раскрывающемся списке **Main Form** на странице **Form**, после чего Delphi автоматически вносит соответствующие изменения в файл проекта.

Типичная форма представляет собой прямоугольное окно с рамкой (рис. 4.1). Большинство окон содержит *область заголовка*, в которой расположены значок заголовка, заголовок и ряд кнопок, позволяющие свертывать, развертывать (восстанавливать прежние размеры и положение) и закрывать окно, вызывать окно подсказки. Во многих формах отображаются также строка главного меню (под областью заголовка) и строка состояния (обычно в нижней части окна). При необходимости в форме могут автоматически появляться полосы прокрутки, предназначенные для просмотра содержимого окна. Остальная часть пространства окна называется *клиентской областью*. В ней можно размещать элементы управления, выводить текст и графику, манипулировать дочерними окнами.

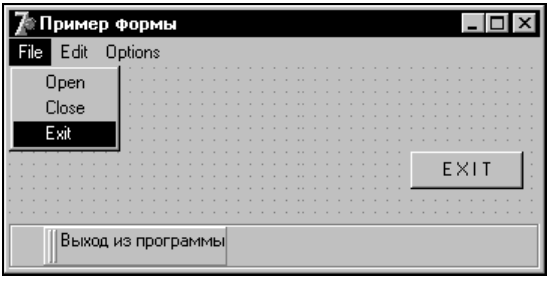


Рис. 4.1. Вид стандартной формы

Форма может быть модальной и немодальной. *Немодальная* форма позволяет без ее закрытия переключиться в другую форму приложения. *Модальная* форма требует обязательного закрытия перед обращением к любой другой форме приложения.

Формы, которые отображают различные сообщения и требуют от пользователя ввода какой-либо информации, часто называют *диалоговыми окнами*. В свою очередь, диалоговое окно также может быть немодальным или модальным.

В Windows есть два основных типа приложений: однодокументные, или SDI (Single Document Interface — однодокументный интерфейс), и многодокументные, или MDI (Multiple Document Interface — многодокументный интерфейс). *Однодокументные* приложения состоят из одной или нескольких независимых друг от друга форм. В SDI-приложении ни одно окно на экране визуально не содержит в себе другие окна, поэтому иногда неясно, какое из них является главным (родительским) окном приложения. В *многодокументном* приложении главное окно содержит дочерние окна, размещаемые в его пределах. Особенности MDI-приложений будут рассмотрены позже.

Характеристики формы

Как и любой другой визуальный компонент, форма имеет свойства, методы и события, общие для всех визуальных компонентов, многие из которых уже были рассмотрены в *главе 3*, посвященной интерфейсным элементам. Наряду с ними у формы есть и специ-

фические свойства, методы и события, определяемые ее особым значением. Некоторые из них характеризуют форму как *главный объект приложения*, скажем, свойство `BorderIcons`, другие присущи форме как контейнеру других компонентов, например, свойства `AutoScroll` и `ActiveControl`.

Система Delphi при добавлении новой формы в проект автоматически создает один экземпляр класса (`Form1`, `Form2` и т. д.), внося соответствующие изменения в файл проекта, например, добавляя строку кода:

```
Application.CreateForm(TForm1, Form1);
```

Управлять процессом автоматического создания форм можно, непосредственно редактируя файл проекта (не рекомендуется делать неопытным программистам) или выполняя настройки в окне параметров проекта (список **Auto-create forms** на странице **Form**) (см. *главу 1*, рис. 1.13). Если форма переведена из этого списка в список **Available forms** доступных форм проекта, то инструкция ее создания исключается из файла проекта, и программист в ходе выполнения приложения должен динамически создать экземпляр этой формы.

Для *создания экземпляров форм* служит метод (конструктор) `Create`. Сам класс формы обычно предварительно описывается при конструировании приложения, и для формы уже существуют файлы формы (dfm) и программного модуля (pas).

Например, в процедуре

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // Форма создается, но не отображается на экране
  Form2 := TForm2.Create(Application);
  Form2.Caption := 'Новая форма';
end;
```

создается форма `Form2`, принадлежащая объекту приложения и имеющая заголовок `Новая форма`.

При создании и использовании формы генерируются следующие события типа `TNotifyEvent`, указанные в порядке их возникновения:

- `OnCreate`;
- `OnShow`;
- `OnResize`;
- `OnActivate`;
- `OnPaint`.

Событие `OnCreate` генерируется только один раз — при создании формы, остальные события происходят при каждом отображении, активизации и каждой прорисовке формы соответственно.

В обработчик события `OnCreate` обычно включается код, устанавливающий начальные значения свойств формы, а также ее элементов управления, т. е. выполняющий начальную инициализацию формы в дополнение к установленным на этапе разработки приложения параметрам. Кроме того, в обработчик включаются дополнительные опера-

ции, которые должны происходить однократно при создании формы, например, чтение из файла некоторой информации и загрузка ее в список.

Приведем в качестве примера процедуру, обрабатывающую событие `OnCreate` формы `Form2`:

```
procedure TForm2.FormCreate(Sender :TObject);
begin
  Form2.Caption := 'Пример формы';
  Edit1.SetFocus;
  ComboBox2.Items.LoadFromFile('list.txt');
  Button3.Enabled := False;
end;
```

При создании форма получает новый заголовок `Пример формы`, в комбинированный список `ComboBox2` загружаются данные из файла list.txt, кнопка `Button3` блокируется, а фокус ввода устанавливается на редактор `Edit1`.

Из всех созданных форм Delphi при выполнении приложения автоматически делает видимой главную форму, для этого свойство `Visible` этой формы устанавливается в значение `True`. Для остальных форм значение данного свойства по умолчанию равно `False`, и после запуска приложения они на экране не отображаются. Если формы создаются вручную, то их отображение и скрытие в процессе работы приложения регулируется программистом через свойство `Visible`. Даже если форма невидима, ее компонентами можно управлять, например, из других форм.

Замечание

Дочерние формы многодокументного приложения становятся видимыми на экране сразу после их создания.

Например, в процедурах

```
procedure TForm1.btnShowForm2Click(Sender :TObject);
begin
  Form2.Visible := true;
end;

procedure TForm1.btnHideForm2Click(Sender :TObject);
begin
  Form2.Visible := False;
end;
```

нажатие кнопок `btnShowForm2` и `btnHideForm2`, расположенных в форме `Form1`, приводит, соответственно, к отображению и скрытию формы `Form2`.

Управлять видимостью форм на экране можно также с помощью методов `Show` и `Hide`. Процедура `Show` отображает форму в *немодальном* режиме, при этом свойство `Visible` устанавливается в значение `True`, а сама форма переводится на передний план. Процедура `Hide` скрывает форму, устанавливая ее свойство `Visible` в значение `False`.

Если окно видимо, то вызов метода `Show` переводит форму на передний план и передает ей фокус ввода.

Пример отображения и скрытия формы:

```
procedure TForm1.btnShowForm3Click(Sender: TObject);
begin
  Form3.Show;
end;

procedure TForm1.btnHideForm3Click(Sender: TObject);
begin
  Form3.Hide;
end;
```

Здесь нажатие кнопок `btnShowForm3` и `btnHideForm3`, расположенных в форме `Form1`, приводит соответственно к отображению на экране и удалению с экрана формы `Form3`.

В момент отображения формы на экране ее свойство `Visible` принимает значение `True`, и возникает событие `OnShow`. Соответственно при скрытии формы свойство `Visible` принимает значение `False`, и возбуждается событие `OnHide`.

При получении формой фокуса ввода, например при нажатии кнопки мыши в области формы, происходит ее активизация и возникает событие `OnActivate`, а при потере фокуса — событие `OnDeActivate`.

Событие `OnPaint` генерируется при необходимости *перерисовки* формы, например, при активизации формы, если до этого часть ее была закрыта другими окнами.

Для закрытия формы используется метод `Close`, который, если это возможно, удаляет ее с экрана. В случае закрытия главной формы прекращается работа всего приложения.

В процедуре

```
procedure TForm2.btnCloseClick(Sender: TObject);
begin
  Form2.Close;
end;
```

кнопка `btnClose` закрывает форму `Form2`. Форма делается невидимой, но не уничтожается. Для этой кнопки полезно задать соответствующий заголовок (свойство `Caption`), например, `Закрыть`.

Процедура `Close` не уничтожает созданный экземпляр формы, и форма может быть снова вызвана на экран, в частности, с помощью методов `Show` или `ShowModal`.

Уничтожение формы происходит с помощью методов `Release`, `Free` или `Destroy`, после чего работа с этой формой становится невозможна, и любая попытка обратиться к ней или ее компонентам вызовет исключение (ошибку). Необходимость уничтожения формы может возникнуть при оформлении заставок или при разработке больших приложений, требующих экономии оперативной памяти. Предпочтительным методом удаления формы считается метод `Free`, поскольку он предварительно проверяет возможность удаления. Например, в процедуре

```
procedure TForm3.btnDestroyClick(Sender: TObject);
begin
  Form3.Free;
end;
```

кнопка btnDestroy уничтожает форму Form3. Для этой кнопки полезно задать соответствующий заголовок, например Удалить.

При закрытии и уничтожении формы генерируются следующие события, указанные в порядке их возникновения:

- OnCloseQuery;
- OnClose;
- OnDeActivate;
- OnHide;
- OnDestroy.

Событие OnCloseQuery типа TCloseQueryEvent возникает в ответ на попытку закрытия формы. Обработчик события получает логическую переменную-признак CanClose, определяющую, может ли быть закрыта данная форма. По умолчанию эта переменная имеет значение True, и форму можно закрыть. Если установить параметр CanClose в значение False, то форма остается открытой. Такую возможность стоит использовать, например, для подтверждения закрытия окна или проверки, сохранена ли редактируемая информация на диске. Событие OnCloseQuery вызывается всегда, независимо от способа закрытия формы.

Приведем в качестве примера процедуру закрытия формы:

```
procedure TForm2.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  CanClose := MessageDlg('Вы хотите закрыть форму?', mtConfirmation,
                         [mbYes, mbNo], 0) = mrYes;
end;
```

Здесь при закрытии формы Form2 выдается запрос на подтверждение операции, который представляет собой модальное диалоговое окно с текстом и двумя кнопками — **Yes** и **No**. Нажатие кнопки **Yes** вызывает закрытие формы, при нажатии кнопки **No** закрытие формы не происходит.

Событие OnClose типа TCloseEvent возникает непосредственно перед закрытием формы. Обычно оно используется для изменения стандартного поведения формы при закрытии. Для этого обработчику события передается переменная Action типа TCloseAction, которая может принимать следующие значения:

- caNone (форму закрыть нельзя);
- caHide (форма делается невидимой);
- caFree (форма уничтожается, а связанная с ней память освобождается);
- caMinimize (окно формы сворачивается) — значение по умолчанию для MDI-форм.

При закрытии окна методом Close переменная Action по умолчанию получает значение caHide, и форма делается невидимой. При уничтожении формы, например, методом Destroy, переменная Action по умолчанию получает значение caFree, и форма уничтожается.

Событие OnClose возникает при закрытии формы щелчком мыши на кнопке закрытия системного меню или при вызове метода Close. Когда закрывается главная форма приложения, все остальные окна закрываются без вызова события OnClose.

Так, в процедуре

```
procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if Memo1.Modified then Action:=caNone else Action := caHide;
end;
```

при закрытии формы `Form2` проверяется признак модификации содержимого редактора `Memo1`. Если информация в `Memo1` была изменена, то форма не закрывается.

Событие `OnDestroy` типа `TNotifyEvent` возникает непосредственно перед *уничтожением* формы и обычно используется для освобождения ресурсов.

При каждом *изменении размеров* формы в процессе выполнения приложения возникает событие `OnResize` типа `TNotifyEvent`. В обработчике этого события может размещаться код, например, выполняющий изменение положения и размеров элементов управления окна, не имеющих свойства `Align`.

Рассмотрим следующий пример:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  // Установка размеров и положения сетки строк
  StringGrid1.Left := 10;
  StringGrid1.Top := 5;
  StringGrid1.Width := Form1.ClientWidth — 20;
  StringGrid1.Height := Form1.ClientHeight — 15 — Button1.Height;
  // Установка положения кнопки
  Button1.Left := Form1.ClientWidth — 10 — Button1.Width;
  Button1.Top := Form1.ClientHeight — 5 — Button1.Height;
end;
```

В форме `Form1` находятся два компонента: сетка строк `StringGrid1` и кнопка `Button1`. Эти компоненты расположены в форме следующим образом (рис. 4.2):

- сетка `StringGrid1` занимает всю ширину клиентской области формы `Form3`, отступы слева и справа составляют 10 пикселов;
- кнопка `Button1` выровнена по правому краю сетки `StringGrid1`;
- расстояния между сеткой, кнопкой, верхним и нижним краями формы составляют 5 пикселов.

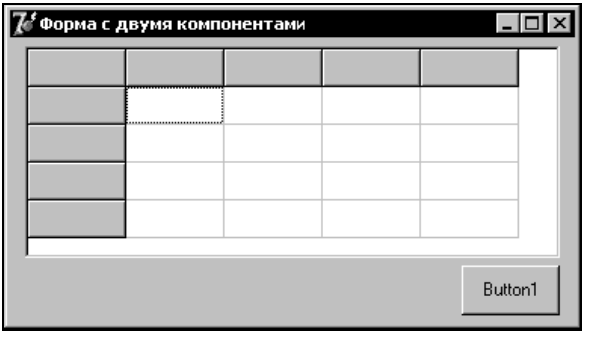


Рис. 4.2. Форма с двумя компонентами

При изменении размеров формы Form1 выполняется пересчет параметров, задающих размеры и положение сетки строк, а также положение кнопки.

Стиль формы определяется свойством FormStyle типа TFormStyle, принимающим следующие значения:

- fsNormal (стандартный стиль, используемый для большинства окон, в том числе и диалоговых);
- fsMDIChild (дочерняя форма в многодокументном приложении);
- fsMDIForm (родительская форма в многодокументном приложении);
- fsStayOnTop (форма, которая после запуска всегда отображается поверх других окон) — обычно используется при выводе системной информации или информационной панели программы.

Форма может изменять стиль динамически — в процессе выполнения программы, например, при выборе пункта меню. При изменении формой стиля возникает событие OnShow.

Пример динамического изменения стиля формы:

```
procedure TForm1.mnuTopClick(Sender: TObject);
begin
  mnuTop.Checked := not mnuTop.Checked;
  if mnuTop.Checked then Form1.FormStyle := fsStayOnTop
                    else Form1.FormStyle := fsNormal;
end;
```

При выборе пункта меню mnuTop форма переключает свой стиль между значениями fsNormal и fsStayOnTop. Смена стиля отображается графически галочкой в заголовке этого пункта меню.

Каждая форма имеет ограничивающую *рамку*. Вид и поведение рамки определяет свойство BorderStyle типа TFormBorderStyle. Оно может принимать следующие значения:

- bsDialog (диалоговая форма);
- bsSingle (форма с неизменяемыми размерами);
- bsNone (форма не имеет видимой рамки и заголовка и не может изменять свои размеры) — часто используется для заставок;
- bsSizeable (обычная форма с изменяемыми размерами) — по умолчанию, имеет строку заголовка и может содержать любой набор кнопок;
- bsToolWindow (форма панели инструментов);
- bsSizeToolWin (форма панели инструментов с изменяемыми размерами).

Замечание

Визуальное отличие между диалоговой и обычной формами заключается в том, что диалоговая форма может содержать в своем заголовке только кнопки закрытия и справки. Кроме того, пользователь не может изменять размеры диалоговой формы.

Невозможность изменения размеров форм некоторых стилей относится только к пользователю — нельзя с помощью мыши передвинуть границу формы в ту или иную сторону. Программно при выполнении приложения для формы любого стиля можно устанавливать любые допустимые размеры окна, а также изменять их.

Пример программного изменения размеров формы:

```
procedure TForm2. btnResizeFormClick(Sender: TObject);
begin
  Form2.Width := Form2.Width + 100;
end;
```

При нажатии кнопки `btnResizeForm` ширина формы `Form2` увеличивается на 100 пикселов, даже если ее свойство `BorderStyle` имеет значение, равное `bsDialog`, `bsSingle` или `bsNone`.

Замечание

Если установить диалоговый стиль формы, то она не становится модальной и позволяет пользователю переходить в другие окна приложения. Для запуска формы, в том числе любой диалоговой, в модальном режиме следует использовать метод `ShowModal`. Таким образом, стиль определяет внешний вид формы, но не ее поведение.

В области заголовка могут отображаться 4 вида кнопок. Реализуемый *набор кнопок* определяет свойство `BorderIcons` типа `TBorderIcons`, которое может принимать комбинации следующих значений:

- `biSystemMenu` (окно имеет системное меню и может содержать кнопки системного меню);
- `biMinimize` (окно содержит кнопку свертывания);
- `biMaximize` (окно содержит кнопку развертывания/восстановления);
- `biHelp` (окно содержит кнопку справки, которая отображает вопросительный знак и вызывает контекстно-зависимую справку).

Системное меню представляет собой набор общих для всех окон Windows команд, например, **Свернуть** или **Закрыть**. При наличии у окна системного меню в области заголовка слева отображается значок приложения, при щелчке на котором и появляются команды этого меню, а в области заголовка справа имеется кнопка закрытия формы (рис. 4.3).

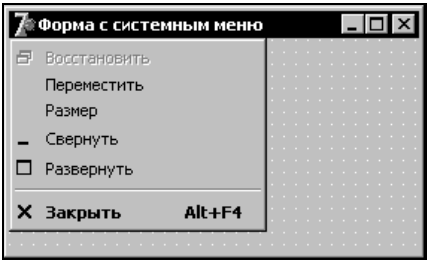


Рис. 4.3. Системное меню формы

Различные значения свойства BorderIcons не являются независимыми друг от друга. Так, если отсутствует системное меню, то ни одна кнопка не отображается. Если имеются кнопки развертывания и свертывания, то не отображается кнопка справки. Возможность появления кнопок также зависит от стиля формы. Например, отображение кнопок развертывания и свертывания возможно только для обычной формы и формы панели инструментов с изменяемыми размерами.

Замечание

Обычно стиль формы и набор кнопок заголовка задаются на этапе разработки приложения в окне Инспектора объектов. При этом в проектируемой форме всегда видны обычная рамка и три кнопки (развертывания, свертывания и закрытия формы), независимо от значения свойств FormStyle и BorderIcons. Заданные стиль формы и набор кнопок становятся видимыми при выполнении программы.

Форма включает в себя клиентскую и неклиентскую области. *Неклиентская* область занята рамкой, заголовком и строкой главного меню. Обычно эта область прорисовывается Windows и программистом не затрагивается. При необходимости изменить отображение в неклиентской области программист может перехватить и обработать сообщение WM_NCPaint.

В *клиентской* области обычно размещаются различные элементы управления, выводится текст или отображается графика. Аналогично тому как свойства Width и Height определяют размеры всей формы, свойства ClientWidth и ClientHeight типа Integer задают ширину и высоту (в пикселах) клиентской части формы. Например, в процедуре

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Caption := 'Клиентская область -' +
                   IntToStr(Form1.ClientWidth) + ' x ' +
                   IntToStr(Form1.ClientHeight);
end;
```

значения размеров клиентской области выводятся в заголовке формы.

Обычно форму перетаскивают мышью, указатель которой устанавливается в любом месте области заголовка. При необходимости можно *переместить форму* и при помещении указателя на ее клиентскую область, для чего требуется описать соответствующие операции программно. Одним из способов является перехват системного сообщения WM_NCHitTest. Для этого создается процедура FormMove, которая анализирует, в каком месте формы находится указатель мыши при нажатии кнопки. Код местоположения указателя мыши содержится в поле Result системного сообщения типа TMessage. Если значение Result равно 1, что соответствует нажатию кнопки мыши в клиентской области, то полю Result присваивается новое значение, равное 2, имитирующее нахождение указателя мыши в области заголовка. В процедуре FormMove первая инструкция inherited осуществляет вызов предопределенного обработчика перехватываемого события.

Чтобы указать среде Delphi, что процедура FormMove является обработчиком события WM_NCHitTest, при ее описании в классе формы TForm1 используется специальный синтаксис, включающий ключевое слово message. Как обработчик системного сообщения, процедура содержит один параметр типа TMessage.

Далее приводится код модуля формы `Form1`, которую можно перемещать мышью, поместив указатель мыши как в область заголовка, так и в клиентскую область.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
  procedure MoveForm(var Msg: TMessage); message WM_NCHitTest;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.MoveForm(var Msg: TMessage);
begin
  inherited;
  if Msg.Result = 1 then Msg.Result := 2;
end;
end.
```

Имена `MoveForm` и `Msg` процедуры и ее параметра могут быть изменены.

Отображаемое формой *меню* задается свойством `Menu` типа `TMainMenu`. При разработке приложения размещение компонента `MainMenu` главного меню в форме вызывает автоматическое присвоение значения `MainMenu1` свойству `Menu`. Это самый простой способ ссылки на главное меню. Если в ходе выполнения приложения какая-либо форма должна отображать различные меню, то через свойство `Menu` можно указать другое главное меню, например, следующим образом:

```
Form1.Menu := MainMenu2;
```

Каждая форма отображает в левой стороне области заголовка свой *значок*, определяемый свойством `Icon` типа `TIcon`. Если форма не является главной в приложении, то этот значок отображается при свертывании формы. Для любой формы свойство `Icon` можно задать с помощью Инспектора объектов или динамически (при выполнении приложения). Если значок не задан, то форма использует значок, указанный в свойстве `Icon` объекта `Application`. Последний выводится также при свертывании и отображении в панели задач Windows значка главной формы приложения.

Например, в процедуре

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Icon.LoadFromFile('Picture1.ico');
end;
```

значок динамически загружается из файла Picture1.ico при создании формы `Form1`.

Размещение и *размер* формы при отображении определяет свойство `Position` типа `TPosition`. Оно может принимать значения, перечисленные далее:

- `poDesigned` (форма отображается в той позиции и с теми размерами, которые были установлены при ее конструировании) — значение по умолчанию. Положение и размеры формы определяются свойствами `Left`, `Top`, `Width` и `Height`. Если приложение запускается на мониторе с более низким разрешением, чем у того, на котором оно разрабатывалось, часть формы может выйти за пределы экрана;
- `poScreenCenter` (форма выводится в центре экрана, ее высота и ширина — свойства `Height` и `Width` — не изменяются);
- `poDefault` (Windows автоматически определяет начальную позицию и размеры формы) — при этом значении программист не имеет возможности управлять этими параметрами, поэтому оно не допускается для форм многодокументных приложений;
- `poDefaultPosOnly` (Windows определяет начальную позицию формы, ее размеры не изменяются);
- `poDefaultSizeOnly` (Windows определяет начальные ширину и высоту формы и помещает форму в позицию, определенную при разработке);
- `PoDesktopCenter` (форма выводится в центре экрана, ее высота и ширина не изменяются);
- `PoMainFormCenter` (форма выводится в центре главной формы приложения, ее высота и ширина не изменяются) — это значение используется для вторичных форм, при применении его для главной формы оно действует как значение `poScreenCenter`;
- `PoOwnerFormCenter` (форма выводится в центре формы, которая является ее владельцем, высота и ширина формы не изменяются) — если для формы не указан владелец (свойство `Owner`), то данное значение аналогично значению `poMainFormCenter`.

Приложение может запоминать расположение и размеры форм и при последующем выполнении правильно отображать формы на экране. Для этого программист должен записать соответствующие данные в инициализационный файл приложения или в системный реестр Windows, а при последующем выполнении приложения считать эти данные и установить их для форм.

Свойство `Active` типа `Boolean` позволяет определить *активность* формы. В любой момент времени активной может быть только одна форма, при этом ее заголовок выделяется особым цветом (обычно синим). Если свойство `Active` имеет значение `True`, то форма активна (находится в фокусе ввода), если `False` — то неактивна. Это свойство доступно для чтения во время выполнения программы. Если требуется активизировать форму программно, следует использовать свойство `WindowState` или метод `Show` (`ShowModal`).

Замечание

В многодокументном приложении родительское окно не может быть активным независимо от цвета заголовка. Для определения активного дочернего окна многодокументного приложения служит свойство `ChildActiveForm` типа `TForm` родительской формы.

В следующем примере

```
procedure TForm1.CheckFormActive(Sender :TObject);
begin
  if Form1.Active then Form1.Caption := '1-я форма активна'
                  else Form1.Caption := '1-я форма неактивна';
  if Form2.Active then Form2.Caption := '2-я форма активна'
                  else Form2.Caption := '2-я форма неактивна';
end;
```

процедура `CheckFormActive` модуля главной формы выполняет проверку активности для двух форм приложения и отображает соответствующую информацию в заголовках форм.

Свойство `WindowState` типа `TWindowState` определяет состояние отображения формы и может принимать одно из трех значений:

- `wsNormal` (обычное состояние) — по умолчанию;
- `wsMinimized` (свернута);
- `wsMaximized` (развернута).

Пример управления состоянием формы:

```
procedure TForm1.btnMiniFormClick(Sender: TObject);
begin
  Form2.WindowState := wsMinimized;
end;

procedure TForm1.btnNormalFormClick(Sender: TObject);
begin
  Form2.WindowState := wsNormal;
end;
```

Кнопки `btnMiniForm` и `btnNormalForm` в форме `Form1` сворачивают и восстанавливают обычное состояние формы `Form2` соответственно.

Форма, для которой изменяется состояние отображения на экране, предварительно должна быть создана методами `CreateForm` или `Create`. Если форма не создана, то при обращении к ней будет сгенерировано исключение, несмотря на то, что переменная формы объявлена в модуле. Если форма создана, но не отображается на экране, то изменения ее состояния (свойства `WindowState`) происходят, однако пользователь не видит этого до тех пор, пока форма не будет отображена на экране.

Будучи контейнером, форма содержит другие элементы управления. Оконные элементы управления (потомки класса `TWinControl`) могут получать фокус ввода. Свойство `ActiveControl` типа `TWinControl` определяет, какой элемент формы находится в фокусе. Для выбора элемента, находящегося в фокусе ввода (активного элемента), можно устанавливать это свойство в нужное значение при выполнении программы:

```
Form1.ActiveControl := Edit2;
```

Эту же операцию выполняет метод `SetFocus`, который устанавливает фокус ввода для оконного элемента управления:

```
Edit2.SetFocus;
```

В случае, когда размеры окна недостаточны для отображения всех содержащихся в форме интерфейсных компонентов, у формы могут появляться *полосы прокрутки*. Свойство `AutoScroll` типа `Boolean` определяет, появляются ли они автоматически. Если свойство `AutoScroll` имеет значение `True` (по умолчанию), то полосы прокрутки появляются и исчезают автоматически, без каких-либо действий программиста. Необходимость в полосах прокрутки может возникнуть, например, в случае, если пользователь уменьшит размеры формы так, что не все элементы управления будут полностью видны. Если же свойство `AutoScroll` установлено в значение `False`, то программист реализует управление просмотром информации вручную через свойства `HorzScrollBar` (горизонтальная прокрутка) и `VertScrollBar` (вертикальная прокрутка) типа `TControlScrollBar` формы.

Для программного *управления полосами прокрутки* можно использовать метод `ScrollInView`. Процедура `ScrollInView(AControl: TControl)` автоматически изменяет позиции полос прокрутки так, чтобы заданный параметром `AControl` элемент управления стал виден в отображаемой области.

Свойство `KeyPreview` типа `Boolean` определяет, будет ли форма *обрабатывать события клавиатуры*, прежде чем их обработают элементы управления формы. Если свойство имеет значение `False` (по умолчанию), то клавиатурные события поступают к активному элементу управления (имеющему фокус ввода). При установке свойства `KeyPreview` в значение `True` форма первой получает сообщения о нажатии клавиш и может на них реагировать, что обычно используется для обработки комбинаций клавиш, независимо от активности элементов управления формы.

Так, в следующей процедуре

```
// Не забудьте установить свойство KeyPreview в значение True
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  MessageDlg('Нажата клавиша ' + Key, mtInformation, [mbOK], 0);
end;
```

форма `Form1` обрабатывает нажатие алфавитно-цифровых клавиш, отображая введенный символ в диалоговом окне **Information**.

Замечание

Форма не может обрабатывать нажатие клавиши <Tab> в связи с ее особым назначением.

У формы имеется ряд свойств и методов, например свойство `MDIChildCount` и метод `Cascade`, предназначенных для организации многодокументных приложений.

Рассмотрим в качестве примера приложение, реализующее цифровые часы, похожие на входящие в состав Windows. Приложение состоит из одной формы (рис. 4.4), в которой размещены надпись `Label1`, таймер `Timer1` и главное меню (компонент `MainMenu1` и соответствующая ему строка меню).

Таймер используется для отсчета времени; его интервал задан равным `1000`, и событие `OnTimer` генерируется один раз в секунду. В обработчике этого события текущее значение времени отображается в надписи `Label1`.

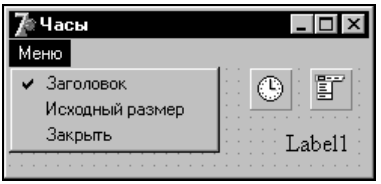


Рис. 4.4. Вид формы при разработке

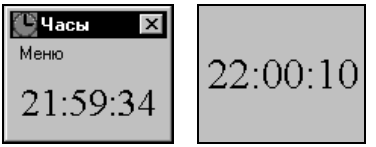


Рис. 4.5. Варианты формы

Меню состоит из трех пунктов **Заголовок** (`mnuCaption`), **Исходный размер** (`mnuInitialSize`) и **Закрыть** (`mnuClose`). В скобках приведены значения свойства `Name` указанных пунктов меню.

Пункт **Заголовок** предназначен для включения и выключения отображения заголовка и меню формы. Первоначально этот пункт отмечен галочкой, и форма отображает заголовок и меню (рис. 4.5, слева). Если выбрать пункт **Заголовок**, то галочка пропадет, а заголовок и меню становятся невидимыми, изменяется также граница окна (рис. 4.5, справа). В этом случае пользователь не сможет изменять размеры окна и перемещать его по экрану. Для восстановления прежнего вида формы нужно сделать двойной щелчок мышью на ней или на надписи `Label1`.

В листинге 4.1 приводится код модуля `uClock` формы `Form1`. Для наглядности многие свойства компонентов получают свои значения в обработчике события `OnCreate`, т. е. при создании формы. В действительности эти действия проще выполнить при разработке приложения с помощью Инспектора объектов.

Листинг 4.1. Код модуля формы приложения, реализующего цифровые часы

```
unit uClock;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls, Menus;
type
  TForm1 = class(TForm)
    Label1:         TLabel;
    Timer1:         TTimer;
    MainMenu1:      TMainMenu;
    mnuMenu:        TMenuItem;
    mnuCaption:     TMenuItem;
    mnuInitialSize: TMenuItem;
    mnuClose:       TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure mnCloseClick(Sender: TObject);
    procedure mnSize0Click(Sender: TObject);
    procedure mnCaptionClick(Sender: TObject);
    procedure FormDblClick(Sender: TObject);
  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

// Установка начальных значений для формы и ее компонентов
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Все эти действия можно выполнить в окне Инспектора объектов
  Application.Icon.LoadFromFile('Clock.ico');
  Form1.FormStyle := fsStayOnTop;
  Form1.Position := poScreenCenter;
  Form1.BorderStyle := bsSizeable;
  Form1.BorderIcons := [biSystemMenu];
  Form1.Constraints.MinWidth := 100;
  Form1.Constraints.MinHeight := 65;
  mnuCaption.Checked := True;
  Form1.ClientWidth := 100;
  Form1.ClientHeight := 45;
  Timer1.Interval := 1000;
  Timer1Timer(Sender);
  // Двойной щелчок на надписи вызывает команду "Заголовок" меню
  Label1.OnDblClick := mnuCaptionClick;
end;

// Изменение размеров надписи Label1 при изменении размеров формы
procedure TForm1.FormResize(Sender: TObject);
begin
  if (Form1.ClientHeight/Form1.ClientWidth) < (Label1.Height/Label1.Width)
    then Label1.Font.Height := Form1.ClientHeight - 2
    else Label1.Font.Height :=
         Round((Form1.ClientWidth-10) * Label1.Height / Label1.Width);
  Label1.Left := (Form1.ClientWidth - Label1.Width) div 2;
  Label1.Top := (Form1.ClientHeight - Label1.Height) div 2;
end;

// Отображение текущего значения времени
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := TimeToStr(Time);
end;

// Отображение и скрытие заголовка и меню формы
procedure TForm1.mnuCaptionClick(Sender: TObject);
begin
  if mnCaption.Checked
  then begin
    Form1.BorderStyle := bsNone;
```

```
    Form1.Menu := Nil;
    end
    else begin
    Form1.BorderStyle := bsSizeable;
    Form1.Menu := MainMenu1;
  end;
  mnuCaption.Checked := not mnuCaption.Checked;
end;

// Восстановление прежнего размера формы
procedure TForm1.mnuInitialSizeClick(Sender: TObject);
begin
  Form1.ClientWidth := 100; Form1.ClientHeight := 45;
  FormResize(Sender);
end;

// Закрытие формы и прекращение работы приложения
procedure TForm1.mnuCloseClick(Sender: TObject);
begin
  Close;
end;

// Активизация пункта "Заголовок" меню необходима,
// когда меню невидимо и должно быть отображено
procedure TForm1.FormDblClick(Sender: TObject);
begin
  mnuCaption.Click;
end;
end.
```

В случае, когда форма отображает заголовок и для нее заданы изменяемые границы, пользователь может варьировать размеры формы. При этом соответственно изменяются размеры надписи `Label1`. Управление размерами надписи осуществляется косвенно, через значение свойства `Height` шрифта текста. Для предотвращения уменьшения формы до размеров, когда шрифт текста надписи будет слишком мелким и плохо различимым, через свойство `Constraints` установлены ограничения на минимальные высоту и ширину формы. Следует учитывать, что это свойство задает минимальные размеры формы, а не ее клиентской области. При выборе пункта меню **Исходный размер** форма восстанавливает размеры клиентской области, заданные при разработке.

Значок приложения и главной формы загружается из файла Clock.ico. В заголовке формы выводятся две кнопки — вызова системного меню и закрытия окна. Форма выводится в центре экрана и расположена поверх всех окон, для этого свойство `FormStyle` установлено в значение `fsStayOnTop`.

Создание стрелочных часов будет рассмотрено в *главе 10*, посвященной графическим возможностям Delphi.

Организация взаимодействия форм

Если одна форма выполняет какие-либо действия с другой формой, то в списке `uses` раздела `implementation` модуля первой формы должна быть ссылка на модуль второй формы.

Для иллюстрации рассмотрим приложение, включающее две формы — `Form1` и `Form2`, для которых имеются модули `Unit1` и `Unit2` соответственно. Далее приводится код модуля `Unit1` первой формы `Form1`.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
// Ссылка на модуль второй формы
uses Unit2;
{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  // Операция со второй формой
  Form2 := TForm2.Create(Self);
end;
end.
```

При нажатии кнопки `Button1` первой формы на экране отображается вторая форма, до этого невидимая. Поскольку операция со второй формой совершается из модуля первой формы, в разделе `implementation` первого модуля помещен код `uses Unit2`.

Ссылку на модуль другой формы можно устанавливать программно, но Delphi позволяет выполнить эту операцию автоматически. Для этого нужно выбрать команду **File | Use Unit** (Файл | Использовать модуль), что приведет к появлению диалогового окна **Use Unit** (рис. 4.6). После выбора нужного модуля и нажатия кнопки **OK** ссылка на него добавляется автоматически.

Если ссылка на требуемый модуль отсутствует, то при компиляции программы появляется диалоговое окно **Information** (рис. 4.7). Оно сообщает, что одна форма использует другую, но модуль второй формы отсутствует в списке `uses` модуля первой формы. Для автоматического добавления ссылки на модуль достаточно нажать кнопку **Yes**.

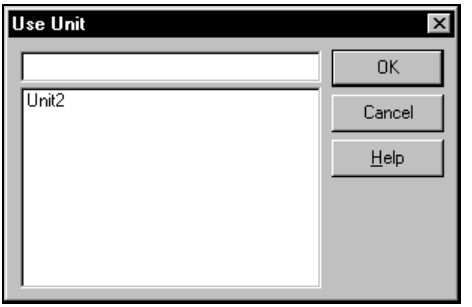


Рис. 4.6. Окно выбора модуля

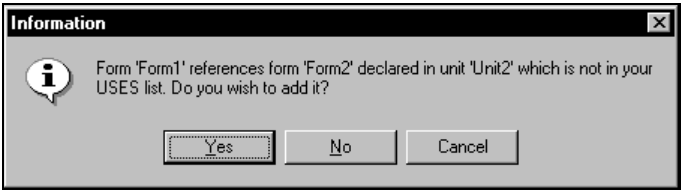


Рис. 4.7. Диалоговое окно **Information**

Форма может выполнять различные операции не только с другой формой, но и с ее отдельными компонентами. В этом случае также нужна ссылка на модуль другой формы. Например:

```
uses Unit2;
...
procedure TForm1.Button2Click(Sender: TObject);
begin
  Label1.Caption := Form2.Edit1.Text;
end;
```

Здесь при нажатии кнопки `Button2` формы `Form1` в надписи `Label1` отображается текст редактора `Edit1`, расположенного в форме `Form2`.

Замечание

Можно выполнять операции с компонентами формы, свернутой или невидимой на экране. Однако в любом случае форма должна быть создана до выполнения любых операций с ней.

Особенности модальных форм

Модальной называется форма, которая должна быть закрыта перед обращением к любой другой форме данного приложения. Если пользователь пытается перейти в другую форму, не закрыв текущую модальную форму, Windows блокирует эту попытку и выдает предупреждающий сигнал. Запрет перехода в другую форму при незакрытой модальной форме относится только к текущему приложению, так что пользователь может активизировать любое другое приложение Windows.

Замечание

Программный доступ к компонентам любой созданной формы приложения возможен, несмотря на наличие в данный момент времени открытой модальной формы.

Модальные формы часто называют *диалоговыми формами (окнами)*, или *диалоговыми панелями*, хотя существуют и немодальные диалоговые окна. Для выполнения различных операций в Windows часто используются стандартные диалоговые формы, с которыми пользователь имеет дело при работе с приложениями. Такие формы называются *общими*, или *стандартными диалоговыми окнами*, для работы с ними Delphi предлагает специальные компоненты. Мы рассмотрим их в *разд. "Стандартные диалоговые окна" далее в этой главе*.

Типичным примером модальной диалоговой формы системы Delphi является диалоговое окно **About Delphi** (рис. 4.8).

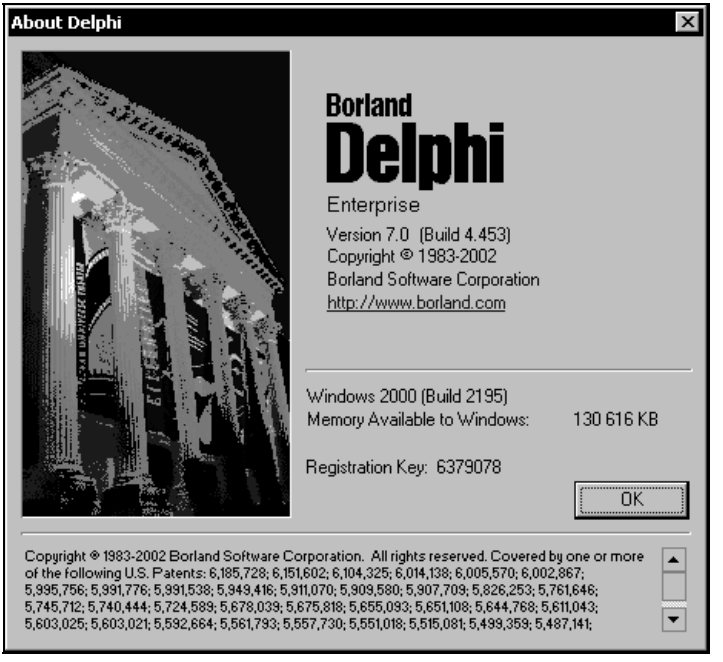


Рис. 4.8. Диалоговое окно **About Delphi**

Диалоговые формы обычно используются при выполнении таких операций, как ввод данных, открытие или сохранение файлов, вывод информации о приложении, установка параметров приложения.

Для *отображения* формы в модальном режиме служит метод `ShowModal`. Например, выполнение процедуры

```
procedure TForm1.mnuAboutClick(Sender: TObject);
begin
  fmAbout.ShowModal;
end;
```

вызываемой путем выбора пункта меню `mnuAbout`, приводит к отображению формы `fmAbout` в модальном режиме. Пункт меню может иметь заголовок (например, **О программе**), задаваемый в свойстве `Caption`. Пользователь может продолжить работу с приложением, только закрыв эту модальную форму.

Многие формы можно отображать и в немодальном режиме, например, так:

```
fmAbout.Show;
```

Напомним, что метод `Show` является процедурой, т. е. не возвращает результат.

При закрытии модальной формы функция `ShowModal` возвращает значение свойства `ModalResult` типа `TModalResult`. Возможные значения этого свойства рассматриваются далее при описании кнопок.

Вообще говоря, код результата при закрытии возвращает любая форма. В данном случае его можно использовать для организации ветвления: возвращаемый после закрытия диалогового окна код результата анализируется и в зависимости от значения выполняются те или иные действия.

В качестве примера рассмотрим, как происходит управление диалоговой формой в реальном приложении.

```
// Процедура находится в модуле формы Form1
procedure TForm1.btnDialogClick(Sender: TObject);
var rez :TModalResult;
begin
  rez := fmDialog.ShowModal;
  if rez = mrOK then
    MessageDlg('Диалог принят.', mtInformation, [mbYes], 0);
  if rez = mrCancel then
    MessageDlg('Диалог отменен.', mtInformation, [mbYes], 0);
end;
...
// Процедуры находятся в модуле формы fmDialog
// Закрытие формы и установка
// кода результата в значение mrOK
procedure TfmDialog.btnCloseClick(Sender: TObject);
begin
  // Порядок инструкций изменять нельзя
  Close;
  ModalResult := mrOK;
end;

// Закрытие формы и установка
// кода результата в значение mrCancel
procedure TfmDialog. btnCancelClick(Sender: TObject);
begin
  // Порядок инструкций изменять нельзя
  Close;
  ModalResult := mrCancel;
end;
```

Кнопка btnDialog формы Form1 открывает диалоговую форму fmDialog. После закрытия диалогового окна кнопкой btnClose или btnCancel анализируется возвращенный код результата и выполняется вывод сообщения на экран.

Как правило, управление кодом результата диалога выполняется не программно (через свойство ModalResult формы), а с помощью кнопок. Чаще всего диалоговая форма (рис. 4.9) содержит кнопки подтверждения и отмены выполненных операций. Кнопка подтверждения диалога в зависимости от назначения может называться по-разному, например: **OK**, **Ввод**, **Открыть**, **Yes**. Кнопка отмены диалога часто называется **Отмена** или **Cancel**.

Как отмечалось, закрыть форму можно, используя свойство ModalResult кнопки. Если свойство имеет значение, отличное от mrNone, то при нажатии кнопки форма автоматически закрывается. При закрытии в качестве результата форма возвращает значение, определяемое свойством ModalResult кнопки, закрывшей эту форму.

Рассмотрим в качестве примера следующую процедуру:

```
procedure TfmDialog.FormCreate(Sender: TObject);
begin
  fmDialog.BorderStyle := bsDialog;
  btnOK.Caption := 'OK';
  btnOK.Default := True;
  btnOK.ModalResult := mrOK;
  btnCancel.Caption := 'Отмена';
  btnCancel.Cancel := True;
  btnCancel.ModalResult := mrCancel;
end;
```

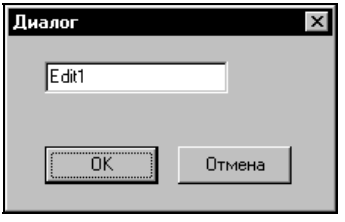


Рис. 4.9. Диалоговая форма с кнопками подтверждения и отмены операций

В ней устанавливаются значения свойств кнопки btnOK подтверждения и кнопки btnCancel отмены диалога fmDialog. При нажатии любой из них форма автоматически закрывается (без выполнения обработчиков события нажатия кнопок) и возвращает соответствующий результат.

Напомним, что обычно свойства кнопок и самой модальной формы задаются в окне Инспектора объектов при проектировании приложения. В приведенном примере для наглядности некоторые свойства устанавливаются в обработчике события OnCreate формы.

Замечание

При закрытии формы методом Close всегда возвращается значение mrCancel ее свойства ModalResult. Скрытие формы методом Hide не изменяет значение свойства ModalResult.

В принципе разработчик может самостоятельно создать любую модальную форму, однако стоит сказать, что для выполнения типовых действий Delphi предлагает ряд предопределенных диалоговых окон. Наиболее простые диалоговые окна реализуются с помощью специальных процедур и функций, в более общих случаях удобно использовать специальные компоненты — стандартные диалоговые окна. Кроме того, ряд диалоговых форм расположен на странице **Dialogs** в Хранилище объектов.

Процедуры и функции, реализующие диалоговые окна

Рассмотрим ряд специальных процедур и функций, предлагаемых Delphi для отображения простых диалоговых окон общего назначения.

Процедура `ShowMessage`, функции `MessageDlg` и `MessageDlgPos` отображают окно (панель) *вывода* сообщений, а функции `InputBox` и `InputQuery` — окно (панель) для *ввода* информации.

Процедура `ShowMessage(const Msg: String)` отображает окно сообщения с кнопкой **OK**. Заголовок содержит название исполняемого файла приложения, а строка `Msg` выводится как текст сообщения.

Рассмотрим отображение простейшего окна сообщения (рис. 4.10) из программы dlgwin.exe.

```
procedure TForm1.btnDialog1Click(Sender: TObject);
begin
  ShowMessage('Простейшее диалоговое окно');
end;
```

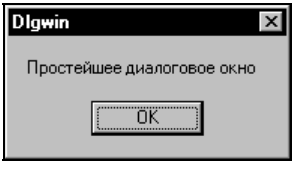


Рис. 4.10. Простейшее окно сообщения

Функция `MessageDlg(const Msg: String; AType: TMsgDlgType; AButtons: TMsgDlgButtons; HelpCtx: Longint): Word` отображает окно сообщения в центре экрана и позволяет получить ответ пользователя. Параметр `Msg` содержит отображаемое сообщение.

Окно сообщения может относиться к различным типам и наряду с сообщением содержать картинки. *Тип окна* сообщения определяется параметром `AType`, который может принимать следующие значения:

- `mtWarning` (окно содержит черный восклицательный знак в желтом треугольнике и заголовок **Warning**);
- `mtError` (окно содержит белый косой крест в красном круге и заголовок **Error**);
- `mtInformation` (окно содержит синюю букву "i" в белом круге и заголовок **Information**);

- `mtConfirmation` (окно содержит синий знак "?" в белом круге и заголовок **Confirmation**);
- `mtCustom` (окно не содержит картинки, в заголовке выводится название исполняемого файла приложения).

Параметр `AButtons` задает набор кнопок окна и может принимать любые комбинации следующих значений:

- `mbYes` (кнопка **Yes**);
- `mbNo` (кнопка **No**);
- `mbOK` (кнопка **OK**);
- `mbCancel` (кнопка **Cancel**);
- `mbHelp` (кнопка **Help**);
- `mbAbort` (кнопка **Abort**);
- `mbRetry` (кнопка **Retry**);
- `mbIgnore` (кнопка **Ignore**);
- `mbAll` (кнопка **All**).

Для значения параметра `AButtons` имеются две константы — `mbYesNoCancel` и `mbOKCancel`, задающие предопределенные наборы кнопок:

- `mbYesNoCancel = [mbYes, mbNo, mbCancel]`
- `mbOKCancel = [mbOK, mbCancel]`

При нажатии любой из указанных кнопок (кроме кнопки **Help**) диалоговое окно закрывается, а результат (свойство `ModalResult`) возвращается функцией `MessageDlg`.

Параметр `HelpCtx` определяет контекст (тему) справки, которая появляется во время отображения диалогового окна при нажатии пользователем клавиши <F1>. Обычно значение этого параметра равно нулю.

Пример использования функции `MessageDlg`:

```
procedure TForm1.btnTestDateClick(Sender: TObject);
var rez :TModalResult;
begin
  if length(edtDate.Text) < 8 then begin
    rez := MessageDlg('Неправильная дата!'+#10#13+'Исправить автоматически?',
                      mtError, [mbOK, mbNo], 0);
    if rez = mrOK then edtDate.Text := DateToStr(Date);
    if rez = mrNo then edtDate.SetFocus;
  end;
end;
```

При нажатии кнопки `btnTestDate` производится простейшая проверка даты. Код даты вводится в поле редактирования `edtDate`, размещенное в форме. Если длина даты меньше допустимой, выдается предупреждение с запросом на автоматическую коррекцию (рис. 4.11). При утвердительном ответе пользователя в поле даты записывается текущая дата, при отрицательном — фокус передается полю ввода даты.

Функция `MessageDlgPos(const Msg: String; AType: TMsgDlgType; AButtons: TMsgDlgButtons; HelpCtx: Longint; X, Y: Integer): Word` отличается от функции `MessageDlg` наличием параметров `X` и `Y`, управляющих *положением окна* на экране.

Функция `InputBox(const ACaption, APrompt, ADefault: String): String` отображает диалоговое окно для *ввода строки* текста. Окно выводится в центре экрана и содержит поле ввода с надписью, а также кнопки **OK** и **Cancel**.

Параметр `ACaption` задает заголовок окна, а параметр `APrompt` содержит поясняющий текст к полю ввода. Параметр `ADefault` определяет строку, возвращаемую функцией при отказе пользователя от ввода информации (нажатие кнопки **Cancel** или клавиши <Esc>).

Пример использования функции `InputBox`:

```
procedure TForm1.btnInputNameClick(Sender: TObject);
Var soname  :string;
begin
  soname := InputBox('Пользователь', 'Введите фамилию', 'Иванов');
end;
```

Приведенная процедура отображает окно запроса на ввод фамилии пользователя (рис. 4.12). По умолчанию предлагается `Иванов`.

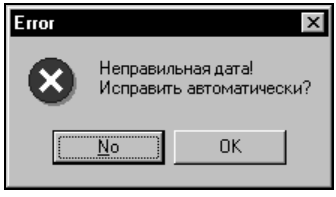


Рис. 4.11. Окно сообщения

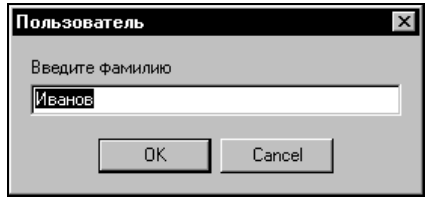


Рис. 4.12. Окно запроса на ввод фамилии

Функция `InputQuery(const ACaption, APrompt: string; var Value: String): Boolean` отличается от функции `InputBox` тем, что вместо третьего параметра — строки по умолчанию — используется параметр `Value`, который в случае подтверждения ввода содержит введенную пользователем строку.

В качестве результата функция возвращает логическое значение, позволяющее определить, каким образом завершен диалог. Если нажата кнопка **OK**, то функция возвращает значение `True`, если нажата кнопка **Cancel** или клавиша <Esc> — значение `False`.

Например, в процедуре

```
procedure TForm1.btnInputNameClick(Sender: TObject);
var soname: string;
begin
  soname := 'Иванов';
  InputQuery('Пользователь', 'Введите фамилию', soname);
end;
```

с помощью функции `InputQuery` выводится окно запроса, аналогичное приведенному на рис. 4.12. Возвращаемый функцией `InputQuery` результат не анализируется.

Кроме рассмотренных диалоговых окон, в Delphi имеется ряд других специализированных диалоговых окон, например диалоговое окно выбора каталога, вызываемое функцией `SelectDirectory` модуля `FileCtrl`.

Стандартные диалоговые окна

В Delphi 7 есть одиннадцать компонентов, находящихся на странице **Dialogs** Палитры компонентов (рис. 4.13) и реализующих диалоговые окна общего назначения. Эти диалоговые окна используются многими Windows-приложениями для выполнения таких операций, как открытие, сохранение и печать файлов, поэтому их часто называют *стандартными*. Например, текстовый процессор Microsoft Word использует большинство из перечисленных далее диалоговых окон. Более того, поскольку стандартные диалоговые окна определяются средой Windows, и мы пользуемся локализованной версией этой операционной системы, диалоговые окна оказываются русифицированными, несмотря на то что сама среда Delphi не локализована.

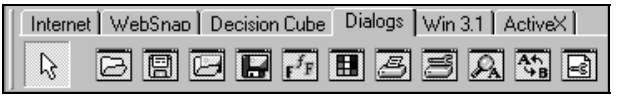


Рис. 4.13. Страница **Dialogs** Палитры компонентов

На странице **Dialogs** Палитры компонентов содержатся следующие компоненты, реализующие стандартные диалоговые окна:

- `OpenDialog` (выбор открываемого файла);
- `SaveDialog` (выбор сохраняемого файла);
- `OpenPictureDialog` (выбор открываемого графического файла);
- `SavePictureDialog` (выбор сохраняемого графического файла);
- `FontDialog` (настройка параметров шрифта);
- `ColorDialog` (выбор цвета);
- `PrintDialog` (вывод на принтер);
- `PrinterSetupDialog` (выбор принтера и настройка его параметров);
- `FindDialog` (ввод строки текста для поиска);
- `ReplaceDialog` (ввод строк текста для поиска и для замены);
- `PageSetupDialog` (установка параметров страницы).

Последнее в приведенном списке диалоговое окно появилось в Delphi 7. Для использования стандартного диалогового окна соответствующий ему компонент должен быть помещен в форму, а его свойства установлены в нужные значения. После этого следует связать вызов диалогового окна с каким-либо событием. Чаще всего таким событием является выбор пункта меню или нажатие кнопки.

Для *вызова* любого стандартного диалогового окна используется метод `Execute` — функция, возвращающая логическое значение. При закрытии диалога кнопкой **OK** (либо **Open** или **Save**) функция `Execute` возвращает значение `True`, а при отмене диалога — значение `False`.

Проверить стандартное диалоговое окно можно уже на этапе разработки приложения. При выборе команды **Test Dialog** (Проверить диалог) контекстного меню или двойном

щелчке на компоненте стандартного диалогового окна оно открывается и работает так же, как и при выполнении приложения.

После закрытия стандартного диалогового окна оно возвращает через свои свойства значения, выбранные или установленные в процессе диалога. Например, при открытии файла возвращаемым значением является имя открываемого файла (`OpenDialog1.FileName`), а при выборе цвета — новый цвет (`ColorDialog1.Color`).

Выбор имени файла

Диалоговые окна выбора имени файла используются в процессах открытия и сохранения файла. Часто эти диалоги называют также диалогами открытия/сохранения файла, хотя на самом деле они позволяют только выбрать имя файла, а все действия по открытию или сохранению файла программируются вручную.

Компонент `OpenDialog` реализует диалог *открытия файла*. При запуске этого диалога появляется окно (рис. 4.14), в котором можно выбрать имя открываемого файла. В случае успешного закрытия диалогового окна (нажатием кнопки **Open**) в качестве результата возвращается выбранное имя файла.

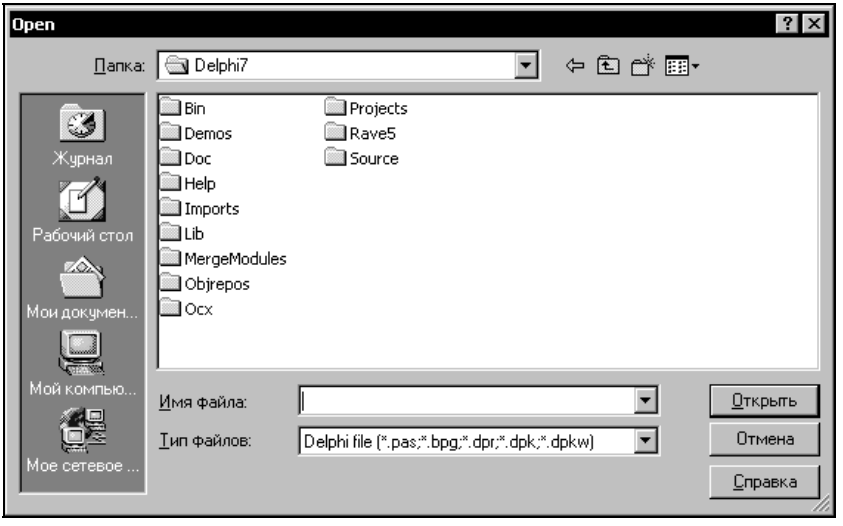


Рис. 4.14. Диалоговое окно открытия файла

Компонент `SaveDialog` предлагает стандартный диалог *сохранения файла*, который отличается от диалога открытия файла только своим заголовком.

Далее перечислены основные свойства компонентов `OpenDialog` и `SaveDialog`.

- `FileName` типа `String` — указывает имя и полный путь файла, выбранного в диалоге. Имя файла отображается в строке редактирования списка **Имя файла** и является результатом диалога.
- `Title` типа `String` — задает заголовок окна. Если свойство `Title` не установлено, то по умолчанию используется заголовок **Open** для `OpenDialog` и заголовок **Save** — для `SaveDialog`.

- `InitialDir` типа `String` — определяет каталог, содержимое которого отображается при вызове диалогового окна. Если каталог не задан, то отображается содержимое текущего каталога.
- `DefaultExt` типа `String` — задает расширение, автоматически используемое в имени файла, если пользователь не указал расширение.
- `Filter` типа `String` — задает маски имен файлов, отображаемых в раскрывающемся списке **Тип файлов**. В диалоговом окне видны имена файлов, совпадающие с указанной маской (на рис. 4.14 это файлы с расширениями txt и doc). По умолчанию значением `Filter` является пустая строка, что соответствует отображению имен файлов всех типов.
- `FilterIndex` типа `Integer` — указывает, какая из масок фильтра отображается в списке. По умолчанию свойство `FilterIndex` имеет значение 1 (используется первая маска).
- `Options` типа `TOpenOptions` — применяется для настройки параметров, управляющих внешним видом и функциональными возможностями диалога. Каждый параметр (флажок) может быть установлен или снят. Свойство `Options` имеет около двух десятков параметров; наиболее важные:
  - `ofAllowMultiSelect` (из списка можно выбрать одновременно более одного файла);
  - `ofCreatePrompt` (при вводе несуществующего имени файла выдается запрос на создание файла);
  - `ofNoLongNames` (имена файлов отображаются как короткие, не более 8 символов для имени и 3 символов для расширения);
  - `ofOldStyleDialog` (создает диалоговое окно в стиле Windows 3.11).

Фильтр представляет собой последовательность значений, разделенных знаком |. Каждое значение состоит из описания и маски, также разделенных знаком |. *Описание* представляет собой обычный текст, поясняющий пользователю данную маску. *Маска* является шаблоном отображаемых файлов и состоит из имени и расширения. Если для одного описания приводится несколько масок, то они разделяются знаком `;`. Например:

```
OpenDialog1.Filter := 'Текстовые файлы|*.txt;*.doc|Все файлы|*.*';
```

Здесь фильтр формируется с двумя масками — маской для текстовых файлов и маской для всех файлов (под текстовыми понимаются файлы типов DOC и TXT).

Так как в раскрывающемся списке диалогового окна отображается только описание фильтра, то для удобства целесообразно включить в описание и маску, например, так:

```
OpenDialog1.Filter:='Текстовые файлы *.txt;*.doc|*.txt;*.doc|Все файлы *.*|*.*';
```

Фильтр обычно формируется при проектировании приложения. Для этого из окна Инспектора объектов щелчком в области значения свойства `Filter` вызывается Редактор фильтра (Filter Editor). Рабочее поле окна Редактора фильтра представляет собой таблицу (рис. 4.15), состоящую из двух столбцов с заголовками **Filter Name** и **Filter**. Зна-

чения фильтра вводятся построчно, при этом в левом столбце указывается описание фильтра, а в правом — соответствующая маска.

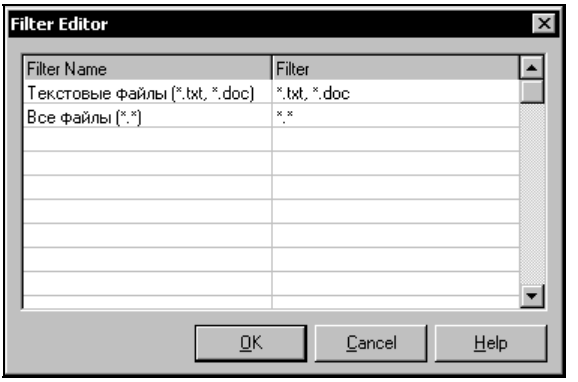


Рис. 4.15. Окно Редактора фильтра

Стандартные диалоговые окна *выбора имени файла* для открытия или сохранения файла вызываются на экран методом `Execute`. Эта функция в качестве результата возвращает логическое значение, позволяющее определить, как закрыт диалог. Если пользователь в процессе диалога нажал кнопку **Open** или **Save**, то диалог считается *принятым*, и функция `Execute` возвращает значение `True`. Если диалог был закрыт любым другим способом, то он считается *не принятым*, и функция возвращает значение `False`.

Рассмотрим пример использования стандартного диалога `OpenDialog`:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
end;
```

При нажатии кнопки `Button2` появляется диалоговое окно `OpenDialog1` выбора имени файла для открытия. При выборе имени текстового файла его содержимое загружается в компонент `Memo1`. Напомним, что многострочный редактор `Memo` позволяет работать с текстами в коде ANSI. При отмене диалога `OpenDialog1` открытие файла не происходит.

Компоненты `OpenPictureDialog` и `SavePictureDialog` вызывают стандартные диалоги *открытия* и *сохранения графических файлов*. Эти стандартные диалоги отличаются от `OpenDialog` и `SaveDialog` видом окон (рис. 4.16) и установленными значениями свойства `Filter`.

Так, по умолчанию свойство `Filter` установлено для отображения графических файлов следующих форматов:

- JPEG (*.jpg);
- JPEG (*.jpeg);
- растровое изображение (*.bmp);
- значок (*.ico);

- метафайл расширенного формата (*.emf);
- метафайл (*.wmf).

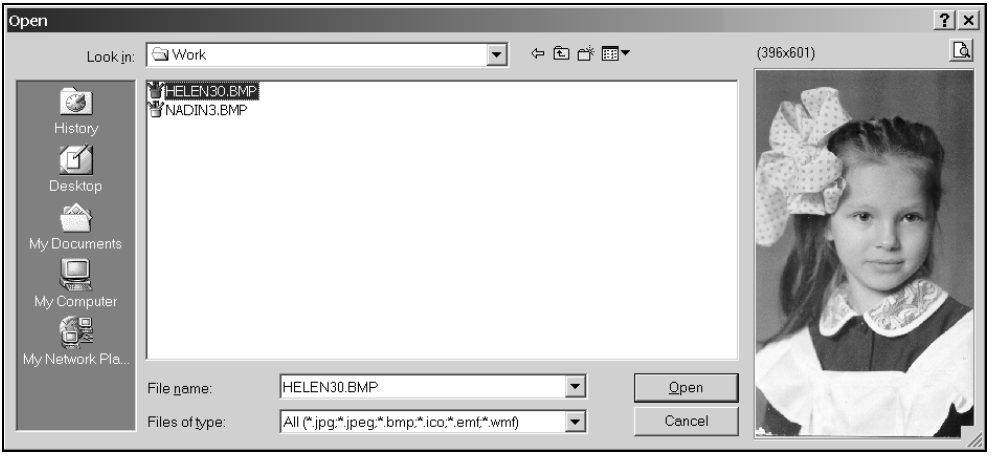


Рис. 4.16. Диалоговое окно открытия графического файла

При использовании диалогов `OpenPictureDialog` и `SavePictureDialog`, а также `OpenDialog` и `SaveDialog` значение свойства `Filter` можно установить ранее рассмотренными способами, а также с помощью функции `GraphicFilter(GraphicClass: TGraphicClass): String`. Параметр `GraphicClass` принадлежит к одному из графических классов: `TBitmap`, `TGraphic` (и его потомки), `TIcon`, `TMetafile` или `TJPEGImage`. Для работы с классом `TJPEGImage` нужно подключить модуль `JPEG`. В качестве результата функция `GraphicFilter()` возвращает строку с фильтрами для указанного графического класса:

- `TBitmap` — `Bitmaps (*.bmp)|*.bmp`;
- `TIcon` — `Icons (*.ico)|*.ico`;
- `TMetafile` — `All (*.emf;*.wmf)|*.emf;*.wmf|Enhanced Metafiles(*.emf)|*.emf|Metafiles(*.wmf)|*.wmf`;
- `TJPEGImage` — `All(*.jpeg;*.jpg)|*.jpeg;*.jpg|JPEG Image File (*.jpeg)|*.jpeg|JPEG Image File (*.jpg)|*.jpg`;
- `TGraphic` — `All(*.jpeg;*.jpg;*.bmp;*.ico;*.emf;*.wmf)|*.jpeg;*.jpg;*.bmp;*.ico;*.emf; *.wmf|JPEG Image File (*.jpeg)|*.jpeg|JPEG Image File (*.jpg)|*.jpg|Bitmaps (*.bmp)|*.bmp|Icons(*.ico) |*.ico|Enhanced Metafiles(*.emf)|*.emf|Metafiles(*.wmf)|*.wmf`.

Замечание

Если модуль `JPEG` в разделе `uses` не указан, то фильтры, соответствующие формату JPEG, будут отсутствовать.

Например, фильтр, заданный как

```
OpenDialog1.Filter := GraphicFilter(TGraphic);
```

позволяет отображать имена графических файлов допустимых форматов.

Выбор параметров шрифта

Диалоговое окно выбора названия (гарнитуры) и других параметров шрифта обеспечивает изменение свойства `Font` (шрифт) для любого визуального компонента, обладающего этим свойством, например, формы или метки (надписи). В Delphi диалог выбора параметров шрифта (рис. 4.17) реализует компонент `FontDialog`.

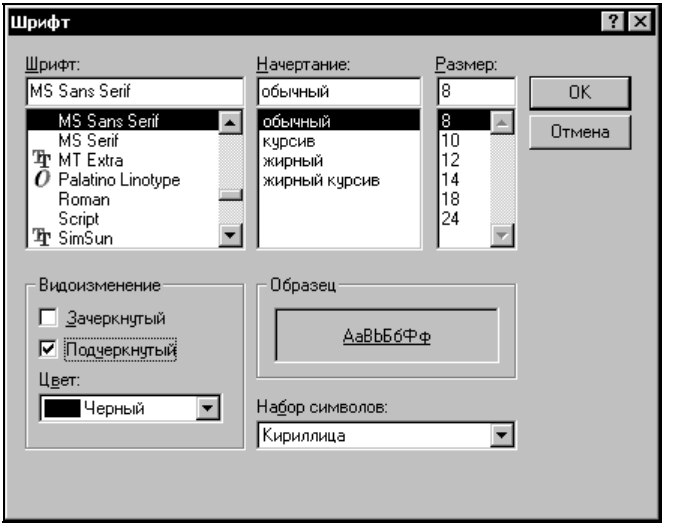


Рис. 4.17. Диалоговое окно выбора параметров шрифта

Основные свойства диалога `FontDialog` перечислены далее.

- Свойство `Font` типа `TFont` определяет параметры шрифта. Управление параметрами шрифта осуществляется через его подсвойства, наиболее важными из которых являются `Name`, `Style`, `Size`, `Color`.
- Свойство `MaxFontSize` типа `Integer` ограничивает доступный в диалоговом окне максимальный размер шрифта. Активно, если включен параметр `fdLimitSize` *(см. далее)*.
- Свойство `MinFontSize` типа `Integer` ограничивает доступный в диалоговом окне минимальный размер шрифта. Активно, если включен параметр `fdLimitSize`.
- Свойство `Device` типа `TFontDialogDevice` указывает тип устройства, для которого устанавливается шрифт, и может принимать одно из трех значений:
  - `fdScreen` (вывод на экран);
  - `fdPrinter` (вывод на принтер);
  - `fdBoth` (вывод на экран и принтер).
- Свойство `Options` типа `TFontDialogOptions` служит для настройки отдельных параметров диалога и включает свыше полутора десятков параметров; важнейшие из них (по умолчанию включен параметр `fdEffects`):
  - `fdEffects` (отображение флажков атрибутов **Подчеркнутый** и **Зачеркнутый**, а также списка **Цвет**);

- `fdLimitSize` (активизация свойств `MaxFontSize` и `MinFontSize`, устанавливающих допустимый диапазон размеров шрифта);
- `fdTrueTypeOnly` (отображение в списке только шрифтов `TrueType`);
- `fdWysiwyg` (отображение в списке шрифтов, одновременно доступных и для экрана, и для принтера).

Так, в приведенной ниже строке кода

```
if FontDialog1.Execute then Label1.Font := FontDialog1.Font;
```

задается шрифт надписи `Label1` с помощью диалога выбора шрифтов.

Выбор цвета

Диалог выбора цвета обеспечивает изменение свойства `Color` для любого визуального компонента, обладающего этим свойством, например, формы или поля редактирования. В Delphi диалог выбора цвета реализует компонент `ColorDialog` (рис. 4.18).

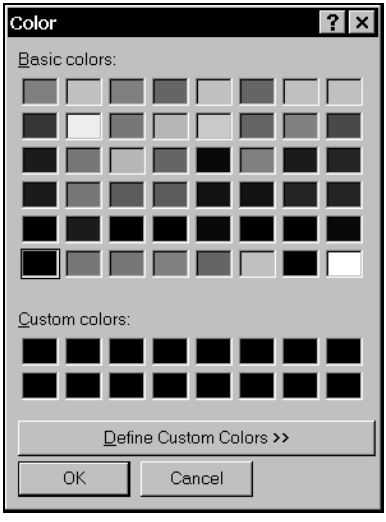


Рис. 4.18. Диалоговое окно выбора цвета

Основными свойствами диалога `ColorDialog` являются:

- `Color` типа `TColor` (определяет выбранный или установленный цвет);
- `Options` типа `TColorDialogOptions` (служит для настройки отдельных параметров диалога); свойство включает следующие параметры:
  - `cdFullOpen` (отображение дополнительной панели выбора цвета);
  - `cdPreventFullOpen` (отключение кнопки **Define Custom Colors >>**);
  - `cdShowHelp` (отображение кнопки **Help**);
  - `cdSolidColor` (задание вместо выбранного цвета ближайшего сплошного цвета);
  - `cdAnyColor` (выбор несплошных цветов).

По умолчанию все параметры выключены.

Например, в следующей строке кода

```
if ColorDialog1.Execute then Edit1.Color := ColorDialog1.Color;
```

с помощью диалога выбора цвета устанавливается цвет редактора `Edit1`.

Замечание

В Delphi на странице **Additional** Палитры компонентов содержится также компонент `ColorBox`. Это специализированный комбинированный список, который позволяет выбрать цвет. Список доступных цветов компонента определяется свойством `Colors[Index: Integer]` типа `TColor`, а текущий (выбранный) цвет указывает свойство `Selected` типа `TColor`.

Выбор принтера и параметров печати

Для организации диалогов, связанных с выводом информации на принтер, предназначены компоненты `PrintDialog` (рис. 4.19) и `PrinterSetupDialog` (рис. 4.20). Они управляют заданиями печати и параметрами принтера соответственно. Эти стандартные диалоговые окна содержат в верхней части группу компонентов, обеспечивающую выбор принтера и настройку его параметров.

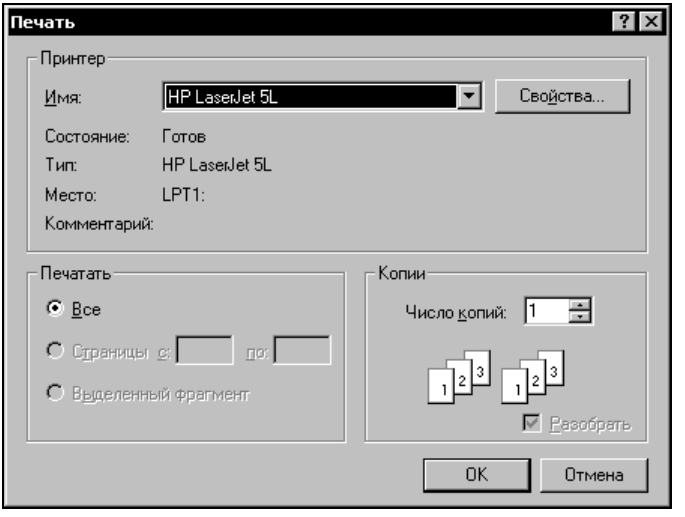


Рис. 4.19. Диалоговое окно управления заданиями печати

Кроме того, диалог `PrintDialog` позволяет при выводе информации на принтер задать ряд таких параметров, как диапазон печатаемых страниц, начальную и конечную страницы, качество печати (разрешение указывается в единицах dpi — точках на дюйм), число копий, возможность печати в файл.

Основные свойства компонента `PrinterDialog`:

- `FromPage` типа `Integer` (номер начальной страницы);
- `ToPage` типа `Integer` (номер конечной страницы);

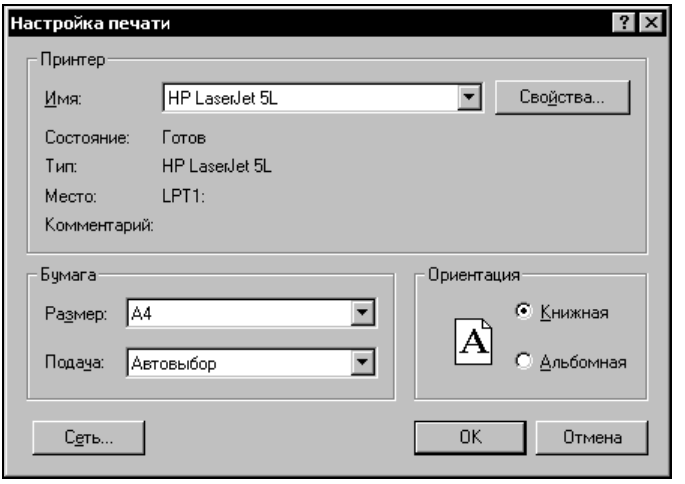


Рис. 4.20. Диалоговое окно настройки параметров принтера

- `Copies` типа `Integer` (число печатаемых копий); если свойство имеет значение 0 или 1, то в окне устанавливается значение 1 (по умолчанию);
- `PrintToFile` типа `Boolean` (состояние флажка **Печать в файл**: если флажок установлен, то свойство `PrintToFile` имеет значение `True`, в противном случае оно имеет значение `False` (по умолчанию));
- `PrintRange` типа `TPrintRange` (диапазон печати):
  - `prAllPages` (печатать все страницы);
  - `prSelection` (печатать выделенный фрагмент);
  - `prPageNums` (печатать указанные страницы с ... по ...);
- `Options` типа `TPrintDialogOptions` (служит для настройки отдельных параметров диалога); содержит следующие параметры:
  - `poDisablePrintToFile` (снять флажок **Печать в файл**) — действует при установленном параметре `poPrintToFile`;
  - `poHelp` (отобразить кнопку **Справка**);
  - `poPageNums` (активизировать переключатель **Страницы**);
  - `poPrintToFile` (отобразить флажок **Печать в файл**);
  - `poSelection` (активизировать переключатель **Выделенный фрагмент**);
  - `poWarning` (выдать предупреждение при попытке послать задание на неинсталлированный принтер).

По умолчанию все параметры выключены.

Замечание

Диалог `PrintDialog` позволяет установить параметры вывода информации на принтер, после чего программист должен организовать собственно процесс печати.

Диалог `PrinterSetupDialog` позволяет устанавливать характеристики бумаги, такие как размер, способ подачи и ориентация. Сделанные в этом диалоговом окне настройки и изменения выполняются Windows автоматически и вступают в силу сразу после закрытия окна, поэтому никаких специфических свойств `PrinterSetupDialog` не имеет. Вызов этого диалогового окна чаще всего выполняется так:

```
PrinterSetupDialog1.Execute;
```

Метод `Execute` вызывается здесь как процедура, а не как функция, поскольку возвращаемый ею результат не используется.

Задание параметров страницы

В Delphi 7 для организации диалога задания параметров страницы введен новый компонент `PageSetupDialog` (рис. 4.21). Он служит для указания размеров бумаги, выбора способа подачи бумаги, выбора ориентации страницы и определения размеров полей.

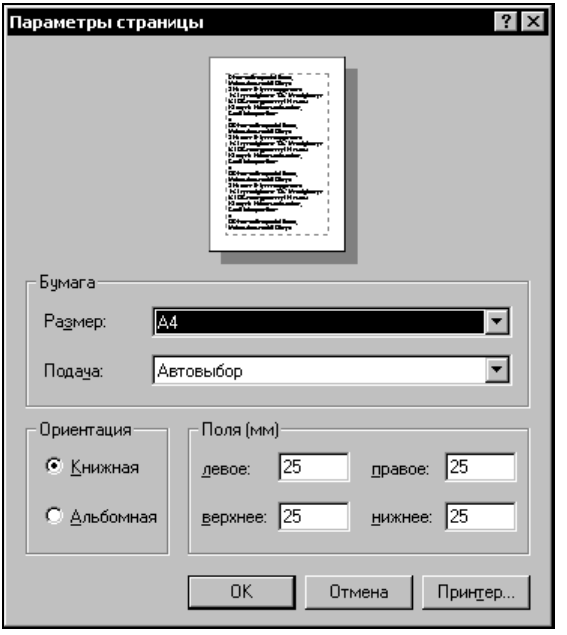


Рис. 4.21. Диалоговое окно задания параметров страницы

К основным свойствам диалога `PageSetupDialog` можно отнести следующие:

- `MarginBottom`, `MarginLeft`, `MarginRight` и `MarginTop` — соответственно определяют нижнее, левое, правое и верхнее поля страницы;
- `PageHeihgt` и `PageWidth` — определяют высоту и ширину страницы;
- `Options` типа `TPageSetupDialogOptions` — используется для настройки ряда параметров диалогового окна (по умолчанию параметр `psoDefaultMinMargins` имеет значе-

ние `True`, остальные параметры имеют значение `False`); содержит такие параметры, как:

- `psoDefaultMinMargins` — запрещает пользователю изменять минимальные поля страницы;
- `psoDefaultDesableMargins` — запрещает пользователю изменять поля страницы;
- `psoDefaultDesableOrientation` — запрещает пользователю изменять ориентацию страницы;
- `psoDefaultDesablePagePainting` — запрещает отображение образца страницы по умолчанию;
- `psoDefaultDesablePaper` — запрещает пользователю изменять размеры страницы и источник бумаги;
- `psoDefaultDesablePrinter` — делает неактивной кнопку вызова диалога настройки дополнительных параметров принтера;

♦ `Units` — служит для выбора единиц измерения размеров полей, задаваемых пользователем (в миллиметрах или в дюймах).

Вызов рассматриваемого диалогового окна выполняется так:

```
PageSetupDialog1.Execute;
```

Метод `Execute` (аналогично предыдущему компоненту) вызывается как процедура, а не как функция, поскольку возвращаемый ею результат не используется.

Ввод строк для поиска и замены

Для организации стандартного диалогового окна **Find** (Найти) (рис. 4.22) предназначен компонент `FindDialog`, который служит для ввода текстовой информации в строку **Find what** (Строка поиска). Введенная строка является значением свойства `FindText` типа `String`.

Кодирование операций, связанных собственно с поиском строки, осуществляется программистом. Для этого используется событие `OnFind` типа `TNotifyEvent`, возникающее при нажатии кнопки **Find Next** (Найти далее). Действия, связанные с поиском текста, должны выполняться внутри процедуры-обработчика этого события. Диалоговое окно **Find** в принципе можно использовать и просто для ввода в программу текстовой информации (значение свойства `FindText`), хотя для этого есть более удобные средства и способы, например функция `InputBox` или однострочный редактор `Edit`.

Рис. 4.22. Диалоговое окно поиска

Параметрами диалога управляет свойство `Options` типа `TFindOptions`, которое может принимать следующие значения (поскольку практически все значения этого параметра для компонента `FindDialog` и компонента `ReplaceDialog`, рассмотренного ниже, аналогичны, здесь приведены и значения, связанные с заменой текста):

- `frDisableMatchCase`, `frDisableWholeWord`, `frDisableUpDown` — снимают флажки **Match case** (Учитывать регистр), **Match whole word only** (Совпадение только по целым словам) и переключатели **Direction** (Направление поиска) соответственно;
- `frDown` — активизирует переключатель **Down** (Вниз) при открытии диалога;
- `frFindNext` — устанавливается при нажатии кнопки **Find Next** и сбрасывается при закрытии диалога;
- `frHideMatchCase`, `frHideWholeWord`, `frHideUpDown` — удаляют из диалога флажки **Match case**, **Match whole word only** и переключатели **Direction** соответственно;
- `frMatchCase` — включается при установке флажка **Match case**;
- `frReplace` (для `ReplaceDialog`) — указывает на необходимость заменить текущее вхождение строки `FindText` строкой `ReplaceText`;
- `frReplaceAll` (для `ReplaceDialog`) — указывает, что необходимо заменить все вхождения строки `FindText` строкой `ReplaceText`;
- `frShowHelp` — отображает кнопку **Help**;
- `frWholeWord` — включается при установке флажка **Match whole word only**.

Рассмотрим в качестве примера следующую процедуру:

```
procedure TForm1.FindDialog1Find(Sender: TObject);
begin
  if pos(FindDialog1.FindText, Memo1.Text) <> 0 then
    begin
      Memo1.HideSelection := False;
      Memo1.SelStart := pos(FindDialog1.FindText, Memo1.Text) - 1;
      Memo1.SelLength := Length(FindDialog1.FindText);
    end
  else MessageDlg('Строка ' + FindDialog1.FindText + ' не найдена!',
                  mtConfirmation, [mbYes], 0);
end;
```

При нажатии кнопки **Find Next** диалога `FindDialog1` в тексте редактора `Memo1` ищется первое появление строки, заданной для поиска (`FindDialog1.FindText`). При успешном поиске найденный текст выделяется, в противном случае выдается сообщение об отсутствии искомой строки. Чтобы найденный текст в компоненте `Memo1` был выделен цветом, перед выделением искомой строки свойству `HideSelection` присваивается значение `False`.

Компонент `ReplaceDialog` организует стандартное диалоговое окно **Replace** (рис. 4.23), предназначенное для ввода текстовой информации в строки **Find what** (Найти) и **Replace with** (Заменить на). Введенные строки являются значениями свойств `FindText` и `ReplaceText` типа `String` соответственно.

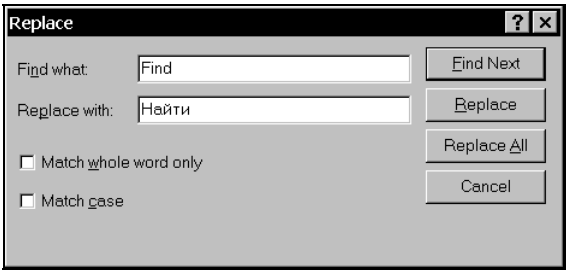


Рис. 4.23. Диалоговое окно поиска и замены текста

Диалоговое окно поиска и замены строк в Delphi реализуется компонентом `ReplaceDialog`, который аналогичен компоненту `FindDialog`, но имеет дополнительно свойство `ReplaceText` типа `String` и событие `OnReplace` типа `TNotifyEvent`, возникающее при нажатии кнопок **Replace** и **Replace All**. Связанные с поиском и заменой текста действия должны выполняться в обработчике этого события.

Реализуемые компонентами `FindDialog` и `ReplaceDialog` диалоговые окна являются немодальными и после их активизации методом `Execute` могут оставаться в неактивном состоянии одновременно с другими окнами. В связи с этим метод `Execute` удобно использовать как процедуру.

Пример текстового редактора

Приведем пример простого текстового редактора, в котором участвуют восемь стандартных диалоговых окон. Вид формы на этапе проектирования показан на рис. 4.24. Для редактирования документа используется компонент `Memo`, расположенный в левой части окна. Восемь кнопок в верхней правой части окна вызывают соответствующие стандартные диалоговые окна, кнопка **Сохранить** служит для записи редактируемого файла на диск, кнопка **Выход** закрывает редактор. В принципе, в приложение можно добавить меню и связать выполняемые при нажатиях кнопок действия с его командами (пунктами).

Рис. 4.24. Форма текстового редактора

Редактор работает с текстовыми файлами, имеющими расширение txt. Изменение шрифта и цвета, выполненные с помощью диалоговых окон, влияют на компонент `Memo`, информация в файле запоминается в символьном коде ANSI.

Код модуля текстового редактора приведен ниже (листинг 4.2).

Листинг 4.2. Код модуля текстового редактора

```
unit uStdDlg;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtDlgs, ExtCtrls, Printers;
type
   TForm1 = class(TForm)
            OpenDialog1: TOpenDialog;
                btnOpen: TButton;
            SaveDialog1: TSaveDialog;
              btnSaveAs: TButton;
               btnClose: TButton;
                  Memo1: TMemo;
                btnSave: TButton;
                btnFont: TButton;
            FontDialog1: TFontDialog;
           ColorDialog1: TColorDialog;
           PrintDialog1: TPrintDialog;
    PrinterSetupDialog1: TPrinterSetupDialog;
            FindDialog1: TFindDialog;
         ReplaceDialog1: TReplaceDialog;
               btnColor: TButton;
               btnPrint: TButton;
        btnPrinterSetup: TButton;
                btnFind: TButton;
             btnReplace: TButton;
    procedure FormCreate(Sender: TObject);
    procedure btnOpenClick(Sender: TObject);
    procedure btnSaveAsClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure btnSaveClick(Sender: TObject);
    procedure btnFontClick(Sender: TObject);
    procedure btnColorClick(Sender: TObject);
    procedure btnPrintClick(Sender: TObject);
    procedure btnPrinterSetupClick(Sender: TObject);
    procedure btnFindClick(Sender: TObject);
    procedure btnReplaceClick(Sender: TObject);
    procedure FindDialog1Find(Sender: TObject);
    procedure ReplaceDialog1Replace(Sender: TObject);

    private
    { Private declarations }

  public
    { Public declarations }
  end;
var Form1          :TForm1;
    EditedFile     :string;
```

```
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  EditedFile := 'NoName.txt';
  Form1.Caption := 'Текстовый редактор ' + ExtractFileName(EditedFile);
  Memo1.HideSelection := False;
  OpenDialog1.Filter:= 'Текстовые файлы *.txt|*.txt|Все файлы *.*|*.*';
  OpenDialog1.DefaultExt := 'TXT';
  SaveDialog1.Filter:='Текстовые файлы *.txt|*.txt|Все файлы *.*|*.*';
  SaveDialog1.DefaultExt := 'TXT';
end;

procedure TForm1.btnOpenClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    EditFile := OpenDialog1.FileName;
    Memo1.Lines.LoadFromFile(EditFile);
    Form1.Caption := 'Текстовый редактор ' + ExtractFileName(EditedFile);
  end;
end;

procedure TForm1.btnSaveClick(Sender: TObject);
begin
  Memo1.Lines.SaveToFile(EditedFile);
  if Memo1.Modified then Memo1.Modified := False;
end;

procedure TForm1.btnSaveAsClick(Sender: TObject);
begin
  if SaveDialog1.Execute then
    begin
      EditFile := SaveDialog1.FileName;
      Memo1.Lines.SaveToFile(EditedFile);
      Form1.Caption := 'Текстовый редактор ' + ExtractFileName(EditedFile);
      if Memo1.Modified then Memo1.Modified := False;
    end;
end;

procedure TForm1.btnFontClick(Sender: TObject);
begin
  if FontDialog1.Execute
    then Memo1.Font := FontDialog1.Font;
end;

procedure TForm1.btnColorClick(Sender: TObject);
begin
  if ColorDialog1.Execute then Memo1.Color := ColorDialog1.Color;
end;
```

```
procedure TForm1.btnPrintClick(Sender: TObject);
var Line      :SystemTextFile;
    i         :integer;
begin
  // Простейший способ печати
  if PrintDialog1.Execute then
    begin
      AssignPrn(Line);
      Rewrite(Line);
      Printer.Canvas.Font := Memo1.Font;
      for I := 0 to Memo1.Lines.Count — 1 do Writeln(Line, Memo1.Lines[i]);
        System.CloseFile(Line);
    end;
end;

procedure TForm1.btnPrinterSetupClick(Sender: TObject);
begin
  PrinterSetupDialog1.Execute;
end;

procedure TForm1.btnFindClick(Sender: TObject);
begin
  FindDialog1.Execute;
end;

procedure TForm1.FindDialog1Find(Sender: TObject);
begin
  // Ищется первое появление в тексте заданной для поиска строки
  if pos(FindDialog1.FindText,Memo1.Text) <> 0 then
    begin
      Memo1.HideSelection := False;
      Memo1.SelStart := pos(FindDialog1.FindText,Memo1.Text) — 1;
      Memo1.SelLength := Length(FindDialog1.FindText);
    end
  else MessageDlg('Строка ' + FindDialog1.FindText + ' не найдена!',
                  mtConfirmation, [mbYes], 0);
end;

procedure TForm1.btnReplaceClick(Sender: TObject);
begin
  ReplaceDialog1.Execute;
end;

procedure TForm1.ReplaceDialog1Replace(Sender: TObject);
label 10;
begin
  // Заменяются все вхождения в текст заданной для поиска строки
  Memo1.HideSelection := True;
10:
  if pos(ReplaceDialog1.FindText,Memo1.Text) <> 0 then
    begin
      Memo1.SelStart := pos(ReplaceDialog1.FindText,Memo1.Text) — 1;
```

```
      Memo1.SelLength := Length(ReplaceDialog1.FindText);
      Memo1.SelText := ReplaceDialog1.ReplaceText;
      goto 10;
    end;
  Memo1.HideSelection := False;
end;

procedure TForm1.btnCloseClick(Sender: TObject);
begin
  Close;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if Memo1.Modified then
    if MessageDlg('Файл ' + ExtractFileName(EditedFile) + ' изменен!' +
                  #10#13+'Подтверждаете выход?',
                  mtConfirmation, [mbYes, mbNo], 0) = mrYes
    then Action := caFree
    else Action := caNone;
end;
end.
```

Перед выходом из программы в обработчике события `OnClose` формы анализируется свойство `Modified` компонента `Memo1`. Если информация была изменена (`Memo1.Modified = True`), то выдается предупреждение с запросом на подтверждение операции. При отмене выхода из программы переменной `Action` присваивается значение `caNone`, и форма не закрывается, в противном случае переменная `Action` получает значение `caFree` (по умолчанию), и форма, а вместе с ней и приложение, закрывается. При сохранении редактируемого файла признак `Memo1.Modified` сбрасывается.

Для улучшения приведенной программы можно вместо командных кнопок использовать меню и панель инструментов.

Шаблоны форм

Хранилище объектов позволяет сохранять формы и другие объекты в качестве шаблонов для последующего использования. *Шаблон* представляет собой своего рода заготовку, настраивая которую можно получить требуемый объект. Различные страницы Хранилища объектов содержат шаблоны следующих форм:

- страница **New**:
  - пустая форма **Form**;
- страница **Forms**:
  - справочное окно **About Box**;
  - форма **Dual List Box** с двумя списками;
  - форма **Tabbed Pages** с блокнотом;

- страница **Dialogs**:
  - диалоговое окно **Dialog with Help** с кнопкой **Help** (два варианта формы, различающиеся расположением кнопок);
  - диалоговое окно **Password Dialog** для ввода пароля;
  - обычное диалоговое окно **Standard Dialog** (два варианта формы, различающиеся расположением кнопок);
  - панель **Reconcile Error Dialog** вывода информации об ошибках;
- страница **IntraWeb** (появилась в Delphi 7):
  - форма приложения **Application Form**;
  - форма страницы **Page Form**.

При добавлении в проект новой формы Delphi автоматически вставляет пустую форму `Form`, к которой разработчик добавляет необходимые интерфейсные компоненты. Новая форма добавляется при выборе команды **File | New | Form** (Файл | Новый | Форма).

Иногда лучше выбрать подходящий к конкретной ситуации шаблон, чем использовать пустую форму. Например, при вводе пароля таким шаблоном является **Password Dialog** (рис. 4.25). Эта форма имеет заголовок `Password Dialog`, имя `PasswordDlg` и содержит надпись `Label1` и две кнопки закрытия диалогового окна — `OKBtn` и `CancelBtn`.

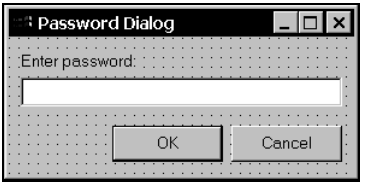


Рис. 4.25. Шаблон диалогового окна **Password Dialog**

Для кнопок закрытия диалогового окна установлены следующие значения свойства `ModalResult`:

- `mrOK` (для `OKBtn`);
- `mrCancel` (для `CancelBtn`).

Программист может изменить значения компонентов, например, заменив английский текст на русский, или расположить в форме новые компоненты.

После этого окно **Password Dialog** можно вызвать на экран с помощью метода `ShowModal`:

```
PasswordDlg.ShowModal;
```

Аналогично можно использовать шаблоны других форм из Хранилища объектов. Кроме того, можно сохранить в Хранилище свои формы, которые вы планируете использовать в других проектах.

В качестве формы, добавляемой к проекту по умолчанию, вместо пустой формы можно задать другую форму из Хранилища объектов. Для этого нужно командой **Tools | Repository** (Средства | Хранилище) открыть окно Хранилища объектов (Object

Repository) и на странице **Forms** выбрать желаемую форму, а также установить для нее флажок **New Form** (рис. 4.26). После закрытия окна выбранная форма будет добавляться в проект по умолчанию. На практике такой подход используется редко ввиду того, что трудно создать форму (кроме пустой), которая подходила бы ко многим случаям.

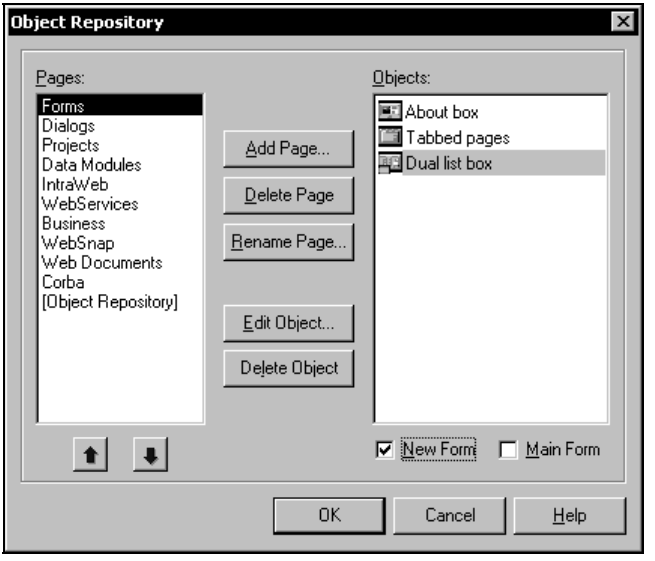


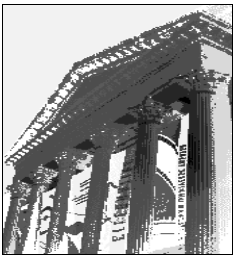
Рис. 4.26. Задание новой формы для вставки в проект по умолчанию

В окне Хранилища объектов можно добавить, переименовать или удалить страницу Хранилища, а также удалить объект или отредактировать информацию об объекте, такую как название или краткое описание.

Кроме шаблонов форм, в Хранилище объектов имеется ряд мастеров (wizards) — специальных программ-утилит, облегчающих создание форм. При этом разработчику предлагается в пошаговом режиме ответить на ряд вопросов, касающихся будущей формы. Программы этого типа достаточно часто используются в Windows в различных ситуациях, например, при построении диаграмм в пакете Microsoft Office или при установке нового оборудования компьютера.

На страницах Хранилища объектов находятся:

- **Business**:
  - мастер **Database Form Wizard** создания формы для работы с базами данных;
  - мастер **TeeChart Wizard** создания диаграмм;
- **ActiveX**:
  - мастер **Active Form Wizard** создания формы для работы с Web-обозревателем.



ГЛАВА 5

Меню, панели инструментов и механизм действий

В этой главе мы рассмотрим меню и панели инструментов, которые являются важнейшими элементами управления пользовательского интерфейса любого приложения. Кроме того, коснемся механизма действий, позволяющего синхронизировать совместную работу различных элементов управления, таких как меню, кнопки и панели инструментов и др.

Меню

Практически все Windows-приложения имеют меню, которое является распространенным элементом пользовательского интерфейса. *Меню* представляет собой список объединенных по функциональному признаку пунктов, каждый из которых обозначает команду или вложенное меню (подменю). Выбор пункта меню равносилен выполнению соответствующей команды или раскрытию подменю.

Обычно в приложении имеется *главное* меню и несколько *контекстных* (*всплывающих* или *локальных*) меню. Главное меню предназначено для управления работой всего приложения, каждое из контекстных меню служит для управления отдельным интерфейсным элементом.

Пункт меню или подменю представляет собой объект типа `TMenuItem`. Отдельный пункт меню (подменю) обычно виден как текстовый заголовок, описывающий назначение пункта меню. Пункт меню может быть выделен (маркирован) для указания включенного состояния. Класс `TMenuItem` служит для представления пунктов главного и контекстных меню. Основные свойства пункта меню:

- `Bitmap` типа `TBitmap` (значок, размещаемый слева от заголовка пункта меню). По умолчанию свойство имеет значение `Nil`, и значок отсутствует;
- `Break` типа `TMenuBreak` (признак разделения меню на столбцы). Свойство `Break` может принимать одно из трех значений:
  - `mbNone` (не разделяется) — по умолчанию;
  - `mbBreak` (пункты меню, начиная с текущего, образуют новый столбец);

  - `mbBreakBar` (пункты меню, начиная с текущего, образуют новый столбец, отделенный линией);

- `Caption` типа `String` (строка текста, отображаемая как заголовок пункта меню). Если в качестве заголовка указать символ "-", то на месте соответствующего пункта меню отображается разделительная линия. При этом, несмотря на отображение линии, свойство `Caption` по-прежнему имеет значение "-";
- `Checked` типа `Boolean` (признак того, помечен пункт или нет). Если свойство установлено в значение `True`, то пункт помечен, и в его заголовке появляется специальная отметка. По умолчанию свойство `Checked` имеет значение `False`, и пункт меню не имеет отметки;
- `AutoCheck` типа `Boolean` (признак автоматического изменения значения свойства `Checked` на противоположное при выборе пользователем пункта меню);
- `Count` типа `Integer` (количество подпунктов в данном пункте меню). Это свойство есть у каждого пункта меню. Если какой-либо пункт не содержит подпунктов, то свойство `Count` имеет нулевое значение;
- `Enabled` типа `Boolean` (признак активности пункта, т. е. будет ли он реагировать на события от клавиатуры и мыши). Если свойство `Enabled` установить в значение `False`, то пункт меню будет неактивным, и его заголовок становится бледным. По умолчанию свойство `Enabled` имеет значение `True`, и пункт меню активен;
- `Items` типа `TMenuItems` (массив подпунктов текущего пункта меню). У каждого пункта меню, имеющего вложенное меню (подменю), пункты этого подменю перечислены в свойстве `Items`. Это свойство позволяет получить доступ к пунктам подменю по их позициям в массиве: `Items[0]`, `Items[1]` и т. д.;
- `RadioItem` типа `Boolean` (вид отметки, появляющейся в заголовке пункта). Если свойство установлено в значение `False` (по умолчанию), то в качестве отметки используется значок "галочка", в противном случае пункт отмечается жирной точкой;
- `ShortCut` типа `TSortCut` (комбинация клавиш для активизации пункта меню). Определить комбинации клавиш можно также с помощью свойства `Caption`, но свойство `ShortCut` предоставляет для этого более широкие возможности. Обозначение комбинаций клавиш, установленных через свойство `ShortCut`, появляется справа от заголовка элемента меню. Наиболее просто задать комбинацию клавиш при конструировании в окне Инспектора объектов, выбрав нужную комбинацию в предлагаемом списке. Кроме того, назначить комбинации клавиш можно с помощью одноименной функции `ShortCut(Key: Word; Shift: TShiftState): TShortCut`. Параметр `Shift` определяет управляющую клавишу, удерживаемую при нажатии алфавитно-цифровой клавиши, указанной параметром `Key`. Если в процессе выполнения программы, например, для пункта меню `mnuTest` требуется задать комбинацию клавиш <Alt>+<T>, то это можно выполнить так:

```
mnuTest.ShortCut := ShortCut(Word('T'), [ssAlt]);
```

- `Visible` типа `Boolean` (признак видимости пункта на экране). Если свойство `Visible` установлено в значение `False`, то пункт меню на экране не отображается. По умолчанию свойство `Visible` имеет значение `True`, и пункт виден в меню.

На рис. 5.1 приведен пример меню **Menu1** (подменю главного меню), разделенного на три столбца. У пункта **Блокировка** этого меню свойство `Break` имеет значение `mbBarBreak`, у пункта **Видимость** свойство `Break` — значение `mbBreak`, для остальных пунктов свойству `Break` установлено значение `mbNone`. Для пункта **Отметка** меню **Menu1** варианты `mbNone` и `mbBreak` значений свойства `Break` в данном случае (при размещении на первой позиции меню) равносильны.

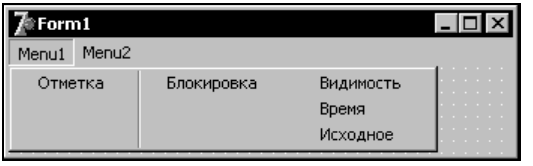


Рис. 5.1. Меню с несколькими столбцами

Основным событием, связанным с пунктом меню, является событие `OnClick`, возникающее при *выборе пункта* с помощью клавиатуры или мыши. В приложении для генерации события `OnClick` или для имитации выбора пункта меню можно использовать метод `Click`. Вызов этой процедуры эквивалентен выбору соответствующего пункта меню пользователем.

Пример имитации выбора пункта меню:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  mnuLockItem.Click;
end;
```

Нажатие кнопки `Button1` приводит к тому же эффекту, что и выбор пункта меню `mnuLockItem`.

Для создания меню (как подменю главного меню) или контекстного меню при разработке приложения используется Конструктор меню. Меню также можно создавать или изменять динамически — непосредственно в ходе выполнения приложения.

Главное меню

Главное меню располагается в верхней части формы под ее заголовком (рис. 5.2) и содержит наиболее общие команды приложения. В Delphi главное меню представлено компонентом `MainMenu`.

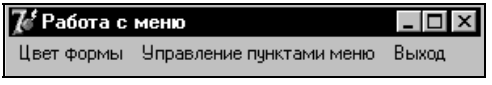


Рис. 5.2. Главное меню

По внешнему виду главное меню представляет собой строку, и его также называют *строчным*. Если пункты меню не умещаются в форме в одну строку, то они переносятся на следующую строку (рис. 5.3). При изменении размеров формы соответствующим образом меняются размеры и размещение пунктов строчного меню. Отметим, что

уменьшение ширины формы ограничено размером самого длинного заголовка, имеющегося в меню.

Рис. 5.3. Главное меню с двумя строками

При проектировании приложения в форме видны компонент `MainMenu` и соответствующая ему строка меню. Отображаемая строка меню выглядит и ведет себя так же, как при выполнении программы. На этапе проектирования приложения для перехода в процедуру обработки события `OnClick` пункта меню следует выбрать этот пункт с помощью клавиатуры или мыши.

Контекстное меню

Контекстное (всплывающее) меню появляется при размещении указателя в форме или в области некоторого элемента управления и нажатии правой кнопки мыши. Обычно контекстное меню содержит команды, влияющие только на тот объект, для которого вызвано это меню, поэтому такое меню также называют *локальным*. На рис. 5.4 показан примерный вид контекстного меню.

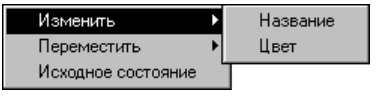


Рис. 5.4. Контекстное меню

Контекстное меню в Delphi представлено компонентом `PopupMenu`. Его основные свойства:

- `AutoPopup` типа `Boolean` (определяет, появляется ли контекстное меню при щелчке правой кнопки мыши и размещении указателя на компоненте, использующем это меню). Если свойство `AutoPopup` имеет значение `True` (по умолчанию), то контекстное меню при щелчке мыши появляется автоматически. Если свойство `AutoPopup` имеет значение `False`, то меню не появляется. Однако в этом случае можно активизировать меню программно, используя метод `Popup`. Процедура `Popup(X, Y: Integer)`, где `X` и `Y` — координаты меню относительно левого верхнего угла экрана монитора, выводит на экран указанное контекстное меню, например:

  ```
  PopupMenu1.Popup(200, 200);
  ```

- `Alignment` типа `TPopupAlignment` (определяет место появления контекстного меню по отношению к указателю мыши). Свойство `Alignment` может принимать следующие значения:

  - `paLeft` (положение указателя определяет положение левого верхнего угла меню) — по умолчанию;

- `paCenter` (положение указателя определяет положение центра меню по горизонтали);
- `paRight` (положение указателя определяет положение правого верхнего угла меню).

Для того чтобы контекстное меню появлялось при щелчке на компоненте, необходимо его свойству `PopupMenu` присвоить в качестве значения имя требуемого контекстного меню.

Пример задания контекстного меню для формы:

```
Form1.PopupMenu := PopupMenu1;
```

Данная инструкция задает для формы `Form1` контекстное меню `PopupMenu1`.

Конструктор меню

Для создания и изменения меню в процессе разработки приложения в среде Delphi предназначен Конструктор меню (Menu Designer). Запуск Конструктора меню выполняется командой **Menu Designer** (Конструктор меню) контекстного меню компонента `MainMenu` или `PopupMenu`, а также с помощью двойного щелчка мыши на этих же компонентах. Предварительно один из этих компонентов следует добавить в форму. Напомним, что компоненты `MainMenu` и `PopupMenu` размещаются на странице **Standard** Палитры компонентов.

Конструктор меню похож на текстовый редактор и предоставляет возможность достаточно просто и удобно конструировать меню любого типа. Меню при конструировании имеет тот же вид, что и при выполнении приложения. Вид меню при конструировании с помощью Конструктора меню показан на рис. 5.5.

При работе с Конструктором меню используются команды его контекстного меню (см. рис. 5.5), вызываемого щелчком правой кнопкой мыши при размещении указателя в области Конструктора меню. С их помощью можно добавить (**Insert**) и удалить (**Delete**) пункт меню, создать подменю (**Create Submenu**), выбрать меню (**Select Menu**), сохранить меню как шаблон (**Save As Template**), вставить меню из шаблона (**Insert From Template**), удалить шаблоны меню (**Delete Templates**) и вставить меню из файла ресурса (**Insert From Resource**).

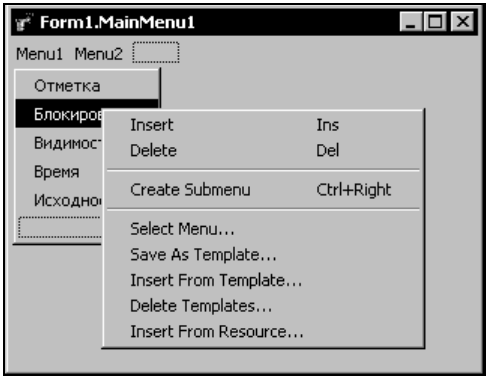


Рис. 5.5. Вид меню при конструировании

При конструировании меню можно также перетаскивать мышью (drag-and-drop) пункты меню и подменю. Используемый совместно с Конструктором меню Инспектор объектов позволяет управлять свойствами отдельных пунктов меню. В частности, заголовок пункта меню задается путем присвоения нужного значения его свойству `Caption`.

В окне Конструктора меню можно сохранить редактируемое меню в качестве шаблона для дальнейшего использования. При выборе команды контекстного меню **Save As Template** (Сохранить как шаблон) появляется окно **Save Template** (Сохранить шаблон) (рис. 5.6). В поле **Template Description** (Описание шаблона) этого окна нужно ввести имя для сохранения шаблона меню. Сохраненный шаблон в дальнейшем можно использовать при создании меню в этом или других приложениях. Список в нижней части окна **Save Template** содержит имена ранее сохраненных шаблонов меню.

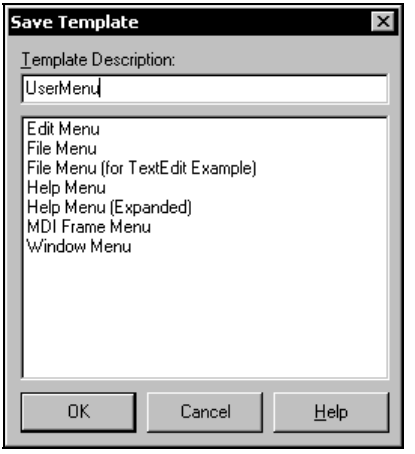


Рис. 5.6. Диалоговое окно **Save Template**

Для загрузки сохраненного шаблона меню следует выполнить команду контекстного меню **Insert From Template** (Вставить из шаблона), при этом появляется окно, подобное приведенному выше. После нажатия кнопки **OK** выбранный в окне шаблон подключается к создаваемому в Конструкторе меню как отдельный пункт.

Меню также можно загрузить из файла ресурса с расширением rc или mnu командой контекстного меню **Insert From Resource** (Вставить из ресурса).

Замечание

С помощью Конструктора меню нельзя сохранить меню в качестве ресурса.

Динамическая настройка меню

Создание и настройка меню с помощью Конструктора меню выполняются при создании приложения. Кроме того, меню можно создавать или изменять динамически непосредственно при выполнении приложения. Например, возможно:

- создать новое меню любого типа или удалить его;
- заблокировать или разблокировать отдельные пункты;

- сделать пункт меню видимым или невидимым;
- добавить или удалить пункт меню;
- изменить название пункта;
- установить или убрать отметку пункта;
- изменить главное меню формы на другое;
- соединить два меню.

Эти возможности обеспечиваются установкой свойствам пунктов меню требуемых значений и вызовом соответствующих методов.

Для *добавления пунктов меню* используются методы `Add` и `Insert`, для *удаления* пунктов меню используется метод `Delete`.

Процедура `Add(Item: TMenuItem)` *добавляет* определяемый параметром `Item` элемент в конец подменю, которое вызвало этот метод. Если подменю не существовало, то оно создается.

Пример добавления пункта меню:

```
procedure Form1.mnuItemAddClick(Sender: TSender);
var NewItem: TMenuItem;
begin
  NewItem := TMenuItem.Create(Self);
  NewItem.Caption := 'Новый элемент';
  mnuFile.Add(NewItem);
end;
```

В этом примере новый пункт добавляется в конец подменю **Файл**. Добавляемый пункт имеет заголовок **Новый элемент**. Предварительно новый пункт создается конструктором `Create`.

Процедура `Insert(Index: Integer; Item: TMenuItem)`, в отличие от предыдущего метода, *добавляет* новый пункт меню в *указанное положение*. Параметр `Index` определяет позицию в массиве элементов меню, в которую вставляется новый пункт. Если значение параметра `Index` выходит за пределы допустимого диапазона, например, больше, чем число элементов подменю модифицируемого пункта меню, то возникает исключение.

Пример добавления пункта меню в заданную позицию:

```
procedure Form1.mnuItemInsertClick(Sender: TSender);
var NewItem: TMenuItem;
begin
  NewItem := TMenuItem.Create(Self);
  NewItem.Caption := 'Второй элемент';
  mnuFile.Insert(2, NewItem);
end;
```

Процедура добавляет новый пункт меню в подменю **Файл**. Новый пункт имеет заголовок **Второй элемент** и добавляется во вторую позицию (отсчет начинается с нуля).

Процедура `Delete(Index: Integer)` *удаляет* указанный пункт меню. Если удаляемый пункт имеет подменю, то оно также удаляется.

Пример удаления пункта меню:

```
procedure Form1.mnuItemDeleteClick(Sender: TSender);
begin
  if mnuFile.Items[2].Caption = 'Второй элемент' then mnuFile.Delete(2);
end;
```

Процедура удаляет пункт меню **Файл**. Удаляемый пункт имеет заголовок **Второй элемент** и находится на второй позиции. Предварительно производится проверка, действительно ли удаляется пункт с нужным названием.

Форма может иметь *больше одного главного меню*. Это используется, например, в случае, когда одно из них содержит заголовки на английском языке, а другое — на русском. Для реализации такой возможности в форму следует поместить два компонента `MainMenu` и подготовить соответствующие меню. После этого при выполнении программы можно подключить к форме любое из этих меню. Для *подключения* к форме главного меню используется свойство `Menu` формы.

Пример переключения между двумя главными меню:

```
if Form1.Menu = EnglishMenu then Form1.Menu := RussianMenu
                            else Form1.Menu := EnglishMenu;
```

Меню имеют имена `EnglishMenu` и `RussianMenu`. Код, выполняющий переключение меню, можно включить в соответствующий обработчик.

Напомним, что `Menu` является одним из свойств формы, указывающим на главное меню, которое в настоящий момент является *активным*.

При организации приложений, содержащих несколько взаимосвязанных форм, может возникнуть необходимость *соединения (слияния)* меню различных форм. Обычно это нужно для меню `MainMenu` главной и подчиненной форм. Для многодокументных приложений соединение меню главной и дочерней формы выполняется автоматически при создании дочерней формы. В однодокументном приложении соединение меню выполняется программно.

Для *соединения* двух строчных меню служит метод `Merge`. Процедура `Merge(Menu: TMainMenu)` соединяет меню, заданное параметром `Menu`, с меню, вызвавшим этот метод. Соединение меню выполняется на уровне пунктов меню, расположенных непосредственно в строке меню. Например, если первое и второе меню выглядят как **File Help** и **File Edit View** соответственно, то результирующее (полученное после соединения) меню будет представлять собой строку вида: **File Edit View Help**.

Соединение меню можно выполнить автоматически при создании форм, установив свойство `AutoMerge` типа `Boolean` в значение `True` для подчиненных (вторичных, или дочерних) форм. Для главной формы приложения свойство `AutoMerge` должно быть установлено в значение `False`. Поскольку в многодокументном приложении соединение выполняется автоматически, свойство `AutoMerge` меню всех форм должно иметь значение `False`.

Порядок следования пунктов меню при соединении определяется значениями свойства `GroupIndex` типа `Byte` отдельных пунктов обоих меню. Если значения свойства `GroupIndex` для обоих пунктов меню равны, то пункт второго меню заменяет пункт пер-

вого меню, для которого был вызван метод `Merge`. Если эти значения не равны, то в объединенном меню пункт с меньшим значением размещается слева от пункта с большим значением. По умолчанию свойство `GroupIndex` имеет нулевое значение для всех пунктов меню.

Такое соединение двух меню обычно применяется при создании многооконных приложений. При вызове из главного окна подчиненного окна, например, окна документа в текстовом процессоре Microsoft Word, пункты меню этого окна "вливаются" в главное меню родительского окна, предоставляя пользователю дополнительные команды и возможности. После закрытия подчиненного окна "лишние" пункты меню главного окна автоматически исчезают.

Для *разъединения* соединенных меню используется метод `UnMerge`. Процедура `UnMerge(Menu: TMainMenu)` разъединяет два меню, соединенных с помощью метода `Merge`. Параметр `Menu` указывает меню, которое было соединено с главным меню и которое необходимо отсоединить.

Рассмотрим пример работы с меню приложения, вид формы которого приведен на рис. 5.7.

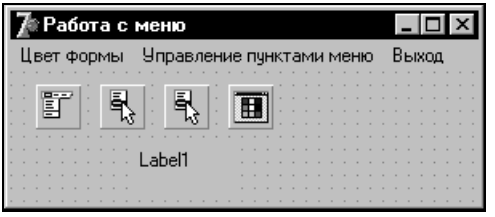


Рис. 5.7. Вид формы при проектировании приложения

Форма приложения содержит надпись `Label1`, главное меню `MainMenu1`, два контекстных меню `PopupMenu1` и `PopupMenu2`, а также стандартный диалог `ColorDialog1` выбора цвета.

Код модуля `uMenu1` формы `Form1` приведен в листинге 5.1.

Листинг 5.1. Пример работы с меню приложения

```
unit uMenu1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, StdCtrls;

type
  TForm1 = class(TForm)
            MainMenu1: TMainMenu;
          mnuFormColor: TMenuItem;
       mnuFormColorRed: TMenuItem;
      mnuFormColorBlue: TMenuItem;
     mnuFormColorReset: TMenuItem;
              mnuItems: TMenuItem;
```

```
      mnuItemChecked: TMenuItem;
         mnuItemLock: TMenuItem;
      mnuItemVisible: TMenuItem;
      mnuItemCaption: TMenuItem;
       mnuItemsReset: TMenuItem;
             mnuExit: TMenuItem;
              Label1: TLabel;
          PopupMenu1: TPopupMenu;
          PopupMenu2: TPopupMenu;
    mnu2FormColorRed: TMenuItem;
   mnu2FormColorBlue: TMenuItem;
  mnu2FormColorReset: TMenuItem;
     mnu3LabelChange: TMenuItem;
    mnu3LabelCaption: TMenuItem;
      mnu3LabelColor: TMenuItem;
       mnu3LabelMove: TMenuItem;
       mnu3LabelLeft: TMenuItem;
      mnu3LabelRight: TMenuItem;
      mnu3LabelReset: TMenuItem;
        ColorDialog1: TColorDialog;

    procedure FormCreate(Sender: TObject);
    procedure mnuFormColorRedClick(Sender: TObject);
    procedure mnuFormColorBlueClick(Sender: TObject);
    procedure mnuFormColorResetClick(Sender: TObject);
    procedure mnuItemCheckedClick(Sender: TObject);
    procedure mnuItemLockClick(Sender: TObject);
    procedure mnuItemVisibleClick(Sender: TObject);
    procedure mnuItemCaptionClick(Sender: TObject);
    procedure mnuItemsResetClick(Sender: TObject);
    procedure mnuExitClick(Sender: TObject);
    procedure mnu2FormColorRedClick(Sender: TObject);
    procedure mnu2FormColorBlueClick(Sender: TObject);
    procedure mnu2FormColorResetClick(Sender: TObject);
    procedure mnu3LabelCaptionClick(Sender: TObject);
    procedure mnu3LabelColorClick(Sender: TObject);
    procedure mnu3LabelLeftClick(Sender: TObject);
    procedure mnu3LabelRightClick(Sender: TObject);
    procedure mnu3LabelResetClick(Sender: TObject);

  private
    { Private declarations }

  public
    { Public declarations }
  end;

var            Form1: TForm1;
     FormColorMemory: longint;
    LabelColorMemory: longint;
     LabelLeftMemory: integer;
```

```
implementation

{$R *.DFM}
// Установка начальных значений параметров
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Задание контекстных меню для формы и надписи
  Form1.PopupMenu := PopupMenu1;
  Label1.PopupMenu := PopupMenu2;
  // Запоминание цветов формы и надписи, установленных при разработке
  FormColorMemory := Form1.Color;
  LabelColorMemory := Label1.Color;
  // Запоминание положения надписи, установленного при разработке
  LabelLeftMemory := Label1.Left;
end;

// Установка красного цвета формы
procedure TForm1.mnuFormColorRedClick(Sender: TObject);
begin
  Form1.Color := clRed;
end;

// Установка синего цвета формы
procedure TForm1.mnuFormColorBlueClick(Sender: TObject);
begin
  Form1.Color := clBlue;
end;

// Восстановление цвета формы, заданного при разработке
procedure TForm1.mnuFormColorResetClick(Sender: TObject);
begin
  Form1.Color := FormColorMemory;
end;

// Установка или снятие отметки в пункте меню
procedure TForm1.mnuItemCheckedClick(Sender: TObject);
begin
  if mnuItemChecked.Checked then mnuItemChecked.Checked := False
                            else mnuItemChecked.Checked := True;
end;

// Блокировка пункта меню
procedure TForm1.mnuItemLockClick(Sender: TObject);
begin
  mnuItemLock.Enabled := False;
end;

// Скрытие пункта меню
procedure TForm1.mnuItemVisibleClick(Sender: TObject);
begin
  mnuItemVisible.Visible := False;
end;
```

```
// Отображение в заголовке пункта меню текущего времени
procedure TForm1.mnuItemCaptionClick(Sender: TObject);
begin
  mnuItemCaption.Caption := 'Время' + TimeToStr(Time);
end;

// Установка пунктов меню в исходное состояние
procedure TForm1.mnuItemsResetClick(Sender: TObject);
begin
  mnuItemChecked.Checked := False;
  mnuItemLock.Enabled    := True;
  mnuItemVisible.Visible := True;
  mnuTime.Caption        := 'Время';
end;

// Закрытие формы
procedure TForm1.mnuExitClick(Sender: TObject);
begin
  Close;
end;

// Имитация выбора пункта mnuFormColorRed главного меню
TForm1.mnu2FormColorRedClick(Sender: TObject);
begin
  mnuFormColorRed.Click;
end;

// Имитация выбора пункта mnuFormColorBlue главного меню
procedure TForm1.mnu2FormColorBlueClick(Sender: TObject);
begin
  mnuFormColorBlue.Click;
end;

// Имитация выбора пункта mnuFormColorReset главного меню
procedure TForm1.mnu2FormColorResetClick(Sender: TObject);
begin
  mnuFormColorReset.Click;
end;

// Изменение заголовка надписи
procedure TForm1.mnu3LabelCaptionClick(Sender: TObject);
begin
  Label1.Caption := InputBox('Изменение заголовка надписи',
                             'Новый заголовок',
                              Label1.Caption);
end;

// Выбор цвета для надписи
procedure TForm1.mnu3LabelColorClick(Sender: TObject);
begin
  if ColorDialog1.Execute then
    Label1.Color := ColorDialog1.Color;
end;
```

```
// Перемещение надписи влево на 10 пикселов
procedure TForm1.mnu3LabelLeftClick(Sender: TObject);
begin
  Label1.Left := Label1.Left — 10;
  if Label1.Left < 10 then mnu3LabelLeft.Enabled := False;
  if not mnu3LabelRight.Enabled then mnu3LabelRight.Enabled := True;
end;

// Перемещение надписи вправо на 10 пикселов
procedure TForm1.mnu3LabelRightClick(Sender: TObject);
begin
  Label1.Left := Label1.Left + 10;
  if Label1.Left > Form1.ClientWidth — Label1.Width — 10
    then mnu3LabelRight.Enabled := False;
  if not mnu3LabelLeft.Enabled then mnu3LabelLeft.Enabled := True;
end;

// Восстановление цвета и положения надписи, заданных при разработке
procedure TForm1.mnu3LabelResetClick(Sender: TObject);
begin
  Label1.Color := LabelColorMemory;
  Label1.Left := LabelLeftMemory;
  mnu3LabelLeft.Enabled := True;
  mnu3LabelRight.Enabled := True;
end;

end.
```

Главное меню имеет следующую структуру (в скобках указаны имена пунктов — значения свойств `Name` элементов):

- **Цвет формы** (`mnuFormColor`):
  - **Красный** (`mnuFormColorRed`);
  - **Синий** (`mnuFormColorBlue`);
  - **Исходный** (`mnuFormColorReset`);
- **Управление пунктами меню** (`mnuItems`):
  - **Отметка** (`mnuItemChecked`);
  - **Блокировка** (`mnuItemLock`);
  - **Видимость** (`mnuItemVisible`);
  - **Название пункта** (`mnuItemCaption`);
  - **Исходное положение** (`mnuItemsReset`);
- **Выход** (`mnuExit`).

Управление цветом формы состоит в его изменении на красный, синий или в восстановлении исходного цвета, запомненного при создании формы. Управление пунктами меню заключается в появлении и устранении в заголовке отметки (**Отметка**), блоки-

ровке и разблокировке (**Блокировка**), скрытии пункта меню и его отображении (**Видимость**), изменении названия (**Название пункта**).

Два компонента — форма `Form1` и надпись `Label1` имеют контекстные меню.

Контекстное меню `PopupMenu1` для формы `Form1` состоит из трех пунктов и по своему содержанию и функциональному назначению дублирует пункт **Цвет формы** главного меню, отличие заключается в именах пунктов (`mnu2FormColorRed`, `mnu2FormColorBlue` и `mnu2FormColorReset`). Обработчики события выбора элемента меню вызывают метод `Click` соответствующего пункта главного меню.

Контекстное меню `PopupMenu2` для надписи `Label1` имеет следующую структуру:

- **Изменить** (`mnu3LabelChange`):
  - **Название надписи** (`mnu3LabelCaption`);
  - **Цвет** (`mnu3LabelColor`);
- **Переместить** (`mnu3LabelMove`):
  - **Влево** (`mnu3LabelLeft`);
  - **Вправо** (`mnu3LabelRight`);
- **Исходное состояние** (`mnu3LabelReset`).

Для изменения названия или цвета надписи открывается диалоговое окно для ввода требуемой информации. При перемещении надпись сдвигается влево или вправо на 10 пикселов. Если надпись приближается к краю формы, то пункт меню, вызывающий движение в этом направлении, блокируется. При удалении от края формы пункт меню разблокируется.

Комбинации клавиш

Комбинации клавиш служат для быстрого вызова часто используемых команд меню. Комбинации клавиш также называют *"горячими"* клавишами, *быстрыми* клавишами или *акселераторами*. Обычно комбинации клавиш задаются при разработке приложения в окне Инспектора объектов. При выполнении приложения также можно задать или изменить комбинации клавиш для отдельных пунктов меню. Комбинации клавиш задаются через свойства `Caption` и `ShortCut`.

Замечание

В отличие от быстрых клавиш (shortcut keys), которые позволяют выполнить команду, не открывая меню, "горячие" клавиши (hot key) используются только при активном (открытом) соответствующем меню.

Для удобной настройки комбинаций клавиш в Delphi имеется специальный компонент `HotKey`, представляющий собой редактор комбинаций клавиш и обеспечивающий возможность изменения комбинации клавиш при выполнении приложения.

Задание комбинаций клавиш для пунктов меню обычно выполняется в диалоговом окне настройки параметров среды приложения. На рис. 5.8 показано диалоговое окно настройки текстового процессора Microsoft Word.

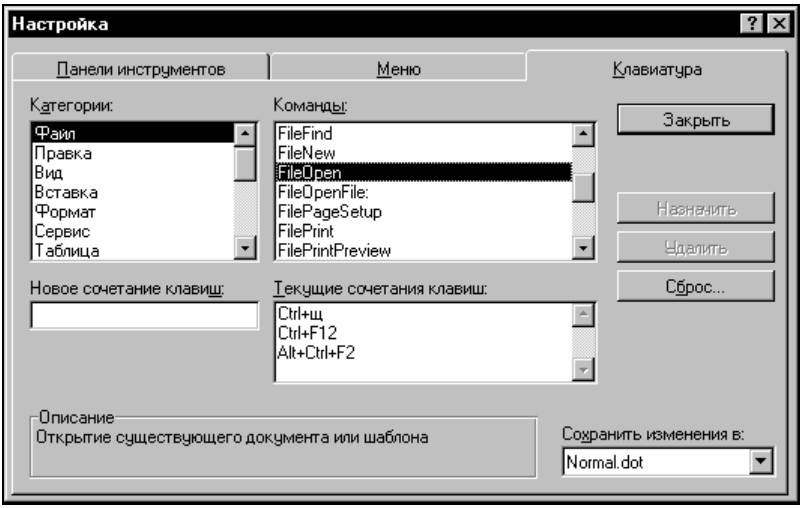


Рис. 5.8. Диалоговое окно настройки текстового процессора

Список с заголовком **Текущие сочетания клавиш** отображает уже выбранные комбинации "горячих" клавиш, а поле с заголовком **Новое сочетание клавиш** позволяет задать новую комбинацию клавиш.

Компонент `HotKey` (рис. 5.9) содержится на странице **Win32** Палитры компонентов и представляет собой редактор, позволяющий задать комбинацию клавиш, которая обычно состоит из *управляющей клавиши* <Alt>, <Ctrl> или <Shift> и другой клавиши (символьной, функциональной или клавиши управления курсором).

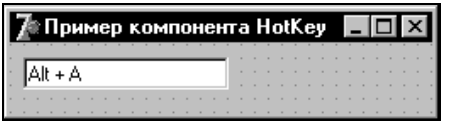


Рис. 5.9. Редактор комбинаций клавиш

Набранную комбинацию клавиш содержит свойство `HotKey` типа `TShortCut`. Значение этого свойства можно присвоить свойству `ShortCut` настраиваемого пункта меню.

Пример изменения комбинации клавиш для пункта меню:

```
procedure TForm1.btnApplyClick(Sender: TObject);
begin
  mnExit.ShortCut := HotKey1.HotKey;
end;
```

При нажатии кнопки `btnApply` пункту `mnExit` назначается новая комбинация клавиш, введенная пользователем в поле компонента `HotKey1`.

Для *управления* возможным *набором комбинаций* клавиш используются свойства `Modifiers` и `InvalidKeys`.

Свойство `Modifiers` типа `THKModifiers` определяет, какие управляющие клавиши используются *по умолчанию* при задании комбинации клавиш.

Это свойство множественного типа и может принимать комбинации следующих значений:

- `hkShift` (в комбинации используется клавиша <Shift>);
- `hkCtrl` (в комбинации используется клавиша <Ctrl>);
- `hkAlt` (в комбинации используется клавиша <Alt>) — по умолчанию;
- `hkExt` (в комбинации нет управляющих клавиш).

В частности, если для свойства `Modifiers` установлена комбинация `[hkCtrl, hkShift]`, то при нажатии пользователем, например, клавиши <P>, в поле редактора `HotKey` будет введена комбинация <Ctrl>+<Shift>+<P>.

Свойство `InvalidKeys` типа `THKInvalidKeys` позволяет указать клавиши, которые при вводе *не будут приняты* редактором `HotKey`. Как и предыдущее, это свойство множественного типа и может принимать комбинации следующих значений:

- `hcNone` (недопустимы немодифицированные клавиши);
- `hcShift` (недопустима клавиша <Shift>);
- `hcCtrl` (недопустима клавиша <Ctrl>);
- `hcAlt` (недопустима клавиша <Alt>);
- `hcShiftCtrl` (недопустима комбинация клавиш <Shift>+<Ctrl>);
- `hcShiftAlt` (недопустима комбинация клавиш <Shift>+<Alt>);
- `hcCtrlAlt` (недопустима комбинация клавиш <Ctrl>+<Alt>);
- `hcShiftCtrlAlt` (недопустима комбинация клавиш <Shift>+<Ctrl>+<Alt>).

По умолчанию свойство `InvalidKeys` имеет значение `[hcNone, hcShift]`.

Настройка системного меню

В ряде случаев для небольших приложений вместо создания строчного меню проще добавить несколько новых команд в системное меню формы. Напомним, что системное меню содержит команды, общие для всех приложений Windows. Вызов системного меню осуществляется щелчком мыши на значке приложения, который находится в левой части заголовка окна.

Для добавления команд нужно изменить системное меню и подготовить обработчики событий выбора новых пунктов этого меню. При изменении системного меню следует иметь в виду, что его владельцем является Windows, а не конкретное приложение.

Для добавления новой команды к системному меню используются API-функции `AppendMenu` или `InsertMenu`, которые добавляют команду в конец меню или в заданную позицию соответственно. Эти функции по своему функциональному назначению не отличаются от рассмотренных методов `Add` и `Insert` компонентов меню.

Функция `InsertMenu(SysMenu: HMenu, uPosition: UINT, uFlags: UINT, uIDNewItem: UINT, lpNewItem: LPCTSTR): Boolean` *добавляет* в системное меню команду `lpNewItem`. Параметр `uIDNewItem` задает уникальный идентификатор (код) нового пункта меню. В случае успешного выполнения функция `InsertMenu` возвращает значение `True`, в противном случае — значение `False`.

Идентификатор `uIDNewItem` задается числом или константой, которые в четырех младших разрядах двоичного представления содержат нули. Это — требование операционной системы Windows, которая использует эти разряды для флагов команды. Проще всего выполнить указанное требование, задавая значение идентификатора в виде шестнадцатеричного числа, у которого младшая цифра равна нулю, например, `$F0`.

Параметр `uFlags` определяет интерпретацию параметра `uPosition`, а также вид и поведение добавляемого пункта. Важнейшие возможные значения параметра `uFlags`:

- `MF_ByCommand` (параметр `uPosition` задает идентификатор нового пункта меню);
- `MF_ByPosition` (параметр `uPosition` определяет позицию нового пункта меню).

Одним из параметров функций `AppendMenu` и `InsertMenu` является *ссылка* `SysMenu` на существующее *системное меню*, поэтому предварительно необходимо получить эту ссылку, для чего удобно использовать функцию `GetSystemMenu`.

Функция `GetSystemMenu(Handle: HWnd, bRevert: boolean): HMenu` возвращает для окна, указанного параметром `Handle`, ссылку на системное меню этого окна. Параметр `bRevert` определяет, что именно возвращается в качестве результата. Если параметр имеет значение `False`, то функция `GetSystemMenu` возвращает ссылку на копию меню, в противном случае — ссылку на само системное меню. Если для настройки используется копия системного меню, то нет необходимости в восстановлении исходного состояния системного меню после завершения работы приложения.

Таким образом, доступ к системному меню осуществляется не с помощью специального компонента, например, `MainMenu`, а через ссылку. При изменении системного меню лучше использовать не само системное меню, а его копию, предоставляемую Windows отдельному приложению. Если приложение работает с самим системным меню, то сделанные в нем изменения отразятся и на других приложениях, использующих это меню.

При выборе стандартных пунктов системного меню, например **Восстановить** или **Закрыть**, Windows автоматически выполняет соответствующие действия. Для обработки событий выбора добавленных (новых) пунктов системного меню необходимо использовать системное сообщение `WM_SysCommand`. Программист должен подготовить обработчик этого сообщения и в нем определить реакцию на выбор нового пункта меню.

В класс формы включается объявление процедуры-обработчика сообщения вида

```
procedure SystemMenu(var Msg: TMessage); message WM_SysCommand;
```

Процедура имеет один параметр типа `TMessage` (или `TWMSysCommand`), который наряду с другой информацией содержит код команды. Для определения того, что этот обработчик вызывается при получении сообщения, используется синтаксис `message WM_SysCommand`. Такое описание означает, что процедура `SystemMenu` обрабатывает событие `WM_SysCommand` выбора пункта системного меню.

В теле процедуры проверяется, какой пункт системного меню выбран, и выполняются соответствующие действия. Чтобы не нарушить нормальное функционирование меню, в теле процедуры должен быть размещен вызов процедуры `Inherited`, которая обеспечивает обработку сообщения по умолчанию.

Рассмотрим пример добавления в четвертую строку системного меню нового пункта **Новый**, при выборе которого будет выводиться простейшее диалоговое окно. Далее приведены фрагменты модуля `uSysMenu`, относящиеся к изменению системного меню формы `Form1`.

```
unit uSysMenu;

interface
...

type
  TForm1 = class(TForm)
...
  // Обработчик системного сообщения WM_SysCommand
  procedure SystemMenu(var Msg: TMessage); message WM_SysCommand;
...
  private
    { Private declarations }
  public
    { Public declarations }
  end;

...

implementation

// Идентификатор нового пункта меню
const IDM_New = $C0;

{$R *.DFM}

procedure TForm1.SystemMenu(var Msg: TMessage);
begin
  // Реакция на новый пункт системного меню
  if Msg.wParam = IDM_New then ShowMessage('Выбран пункт "Новый" ');
  // Обработка сообщения по умолчанию
  inherited;
end;

procedure TForm1.FormCreate(Sender: TObject);
var SM: HMenu;
begin
  // Для настройки используется копия системного меню
  SM := GetSystemMenu(Handle, False);
  // Добавление нового пункта в четвертую позицию системного меню
  InsertMenu(SM, UINT(3), MF_ByPosition, IDM_New, 'Новый');
end;

...
end.
```

Аналогичным способом можно *удалять* и *изменять* отдельные пункты используемого в форме системного меню. Для этого используются API-функции `DeleteMenu` и `ModifyMenu` соответственно.

Пример замены пункта системного меню:

```
const IDM_New2=$D0;
...
// В обработчике сообщения WM_SysCommand необходимо предусмотреть
// реакцию на команду Новый 2
procedure TForm1.FormCreate(Sender: TObject);
var SM: HMenu;
begin
  SM := GetSystemMenu(Handle, False);
  ModifyMenu(SM, UINT(2), MF_ByPosition, IDM_New2, 'Новый 2');
end.
```

Третий пункт системного меню (**Размер**) заменяется пунктом **Новый 2** с идентификатором `IDM_New2`. Параметры функции `ModifyMenu` не отличаются от параметров функции `InsertMenu`.

Панели инструментов

Панель инструментов представляет собой элемент управления, предназначенный для упрощения вызова команд для выполнения наиболее часто используемых операций. Обычно команды, вызываемые с помощью панели инструментов, дублируют часто используемые команды главного или контекстного меню. Панели инструментов содержат такие элементы управления, как кнопки и списки. Во многих Windows-приложениях имеется одна или более панелей инструментов, заметно облегчающих работу пользователя. Например, текстовый процессор Microsoft Word предоставляет восемь предопределенных панелей инструментов, кроме того, при необходимости панели можно перенастроить или даже создать новые. В качестве примера на рис. 5.10 показана панель инструментов **Стандартная** этого текстового процессора.

Рис. 5.10. Панель инструментов **Стандартная** текстового процессора Microsoft Word

Панели инструментов могут быть *статическими*, т. е. постоянно прикрепленными к некоторому краю окна, или *плавающими*, которые допускается перемещать в пределах окна. Создать панель инструментов можно:

- на основе компонента `Panel`;
- на основе специальных компонентов `ToolBar` или `CoolBar`;
- на основе компонента `Form`.

Эти способы подробно рассмотрены в следующих разделах.

Создание панели инструментов на основе компонента *Panel*

Для создания панели инструментов можно разместить в форме компонент `Panel` (панель), расположенный на странице **Standard** Палитры компонентов. Заголовок у этого компонента обычно отсутствует, а выравнивание установлено по одному из краев формы, чаще всего по верхнему. Панель является контейнером для других компонентов, и на ней размещаются различные элементы управления, обычно кнопки быстрого доступа `SpeedButton` и комбинированные списки `ComboBox`.

Если у формы есть главное или контекстное меню, то кнопки панели инструментов (быстрого доступа) обычно дублируют их команды, вызывая обработчики событий `OnClick` "своих" пунктов меню. Так, в процедуре

```
procedure TForm3.sbPrintClick(Sender: TObject);
begin
  mnPrint.Click;
end;
```

кнопка `sbPrint` панели инструментов, предназначенная для печати документа, при нажатии активизирует соответствующий пункт `mnPrint` меню.

На поверхности кнопок панели инструментов размещаются рисунки (свойство `Glyph`) или заголовки (свойство `Caption`), идентифицирующие кнопки и поясняющие их назначение. Например, на кнопку, предназначенную для печати документа, обычно помещают изображение принтера. (Подробная информация, связанная с характеристиками кнопок, в том числе с отображением на них рисунков, приведена в *главе 3*, посвященной основным визуальным компонентам.)

Напомним, что кнопки `SpeedButton` могут образовывать группы зависимых переключателей. Если кнопка должна переключаться независимо от других кнопок, то следует создать группу из одной кнопки `SpeedButton`, при этом ее свойство `AllowAllUp` должно быть установлено в значение `True`.

Рассмотрим пример создания панели инструментов (рис. 5.11) на основе компонента `Panel` с размещением на ней восьми кнопок `SpeedButton` быстрого доступа, комбинированного списка `ComboBox` и однострочного редактора `Edit`.

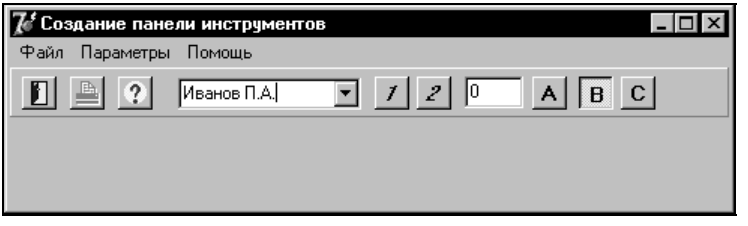


Рис. 5.11. Пример создания панели инструментов

В листинге 5.2 приведен код модуля `uExtend3` с описанием формы `Form3`, содержащей рассматриваемую панель инструментов (фрагменты кода, не связанные с реализацией панели инструментов, исключены).

Листинг 5.2. Пример создания панели инструментов

```
unit uExtend3;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, Buttons, ExtCtrls, StdCtrls;
type
  TForm3 = class(TForm)
    PanelTools: TPanel;
        sbExit: TSpeedButton;
       sbPrint: TSpeedButton;
        sbHelp: TSpeedButton;
     sbRegime1: TSpeedButton;
     sbRegime2: TSpeedButton;
       cbNames: TComboBox;
       edtCode: TEdit;
     sbRegimeA: TSpeedButton;
     sbRegimeB: TSpeedButton;
     sbRegimeC: TSpeedButton;
...
    procedure FormCreate(Sender: TObject);
    procedure sbExitClick(Sender: TObject);
    procedure sbPrintClick(Sender: TObject);
    procedure sbHelpClick(Sender: TObject);
...
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form3: TForm3;
implementation
{$R *.DFM}

// Все операции, кроме загрузки списка, удобнее выполнять через
// Инспектор объектов на этапе создания приложения
procedure TForm3.FormCreate(Sender: TObject);
begin
  PanelTools.Caption := '';
  PanelTools.Align := alTop;
  // Эти три кнопки не могут работать как переключатели
  sbExit.GroupIndex := 0;
  sbPrint.GroupIndex := 0;
  sbHelp.GroupIndex := 0;
  // Эти две кнопки работают как зависимые переключатели
  sbRegime1.GroupIndex := 1;
  sbRegime2.GroupIndex := 1;
  sbRegime1.AllowAllUp := False;
  sbRegime2.AllowAllUp := False;
```

```
  // Эти три кнопки работают как независимые переключатели
  // Вторая кнопка первоначально является нажатой
  sbRegimeA.GroupIndex := 2;
  sbRegimeB.GroupIndex := 3;
  sbRegimeC.GroupIndex := 4;
  sbRegimeA.AllowAllUp := True;
  sbRegimeB.AllowAllUp := True;
  sbRegimeC.AllowAllUp := True;
  sbRegimeB.Down := True;
  // Заполнение списка cbName и поля edtCode
  cbNames.Clear;
  try
    cbNames.Items.LoadFromFile('Personal.txt');
  except
    MessageDlg('Ошибка загрузки списка фамилий!', mtError, [mbOK], 0);
  end;
  edtCode.Text := '0';
  // Блокировка пункта меню и быстрой кнопки, связанных с печатью
  sbPrint.Enabled := False;
  mnuPrint.Enabled := False;
end;

procedure TForm3.sbExitClick(Sender: TObject);
begin
  mnuExit.Click;
end;

procedure TForm3.sbPrintClick(Sender: TObject);
begin
  mnuPrint.Click;
end;

procedure TForm3.sbHelpClick(Sender: TObject);
begin
  mnuHelp.Click;
end;
...
end.
```

Три независимые кнопки дублируют команды меню, выполняющие выход из программы, печать данных и вызов справки. При нажатии любой из этих кнопок выполняются соответствующие команды меню. При запуске приложения, связанные с печатью, пункт меню и быстрая кнопка блокируются.

Две кнопки **1** и **2** образуют группу и работают как зависимые переключатели. Первоначально ни одна из них не выбрана.

Кнопки **A**, **B** и **C** работают как независимые переключатели, для чего каждая из них сгруппирована в отдельную группу. Чтобы эти кнопки включались/выключались щелчком мыши, их свойства `AllowAllUp` установлены в значение `True`. Кнопка **B** первоначально является нажатой.

Комбинированный список содержит фамилии сотрудников организации, из которых выбирается нужная; при отсутствии фамилии в списке ее можно добавить. Фамилии загружаются из файла personal.txt, при этом осуществляются анализ и простейшая обработка возможной ошибки.

Панель инструментов не обязательно выравнивается по краю формы и может находиться в любом месте формы. Она может быть также плавающей и изменять свои размеры. Разработка плавающей панели с изменяемыми размерами — более трудная задача по сравнению с созданием фиксированной панели, т. к. при этом нужно вручную кодировать операции, связанные с перемещением панели и возможной перекомпоновкой ее элементов.

Рассмотрим пример размещения в форме панели (рис. 5.12), с помощью которой можно запускать исполняемые программы (exe-файлы). Для запуска программы достаточно щелкнуть на ее значке.

Рис. 5.12. Панель для запуска программ

Текст модуля `uFmPanel` формы `fmStartFile` приведен в листинге 5.3.

Листинг 5.3. Пример панели для запуска программ

```
unit uFmPanel;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, ComCtrls, Buttons;
type
  TfmStartFile = class(TForm)
    btnDirectory: TButton;
      ImageList1: TImageList;
        btnClose: TButton;
          Panel1: TPanel;

    procedure FileStart(Sender: TObject);
    procedure btnDirectoryClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var fmStartFile :TfmStartFile;
    FileList    :array[1..20] of string;
implementation
uses ShellAPI, CommCtrl, FileCtrl;
{$R *.DFM}

procedure TfmStartFile.FormCreate(Sender: TObject);
begin
  Panel1.Caption := '';
end;

procedure TfmStartFile.FileStart(Sender: TObject);
begin
  WinExec(PChar((Sender as TImage).Hint), SW_ShowNormal);
end;

procedure TfmStartFile.btnDirectoryClick(Sender: TObject);
label 10, 20;
var   SR: TSearchRec;
    Path: string;
       n: integer;
       i: integer;

procedure AddIcon;
begin
  n := n + 1;
  FileList[n] := SR.Name;
  if ExtractIcon(Handle, PChar(FileList[n]), 0) <> 0
    then ImageList_AddIcon(ImageList1.Handle, ExtractIcon (Handle,
                           PChar(FileList[n]), 0))
    else ImageList_AddIcon(ImageList1.Handle, ExtractIcon (Handle,
                           PChar('dos.ico'), 0));
end;

begin
  n := 0;
  // Выбор каталога
  Path := GetCurrentDir;
  // Создание списка программ выбранного каталога
  if SelectDirectory(Path, [], 0) then begin
    if FindFirst(Path+'\*.exe', faArchive, SR) = 0 then begin
       AddIcon;
10:   if FindNext(SR) = 0 then begin
         AddIcon;
         if n = 20 then goto 20 else goto 10;
      end;
    end;
  end;

20:
  FindClose(SR);
```

```
  // Отображение значков программ
  for i := 1 to n do
    with TImage.Create(Self) do begin
      Parent := Panel1;
      Width := ImageList1.Width;
      Height := ImageList1.Height;
      Hint := FileList[i];
      ShowHint := True;
      OnClick := FileStart;
      if i <= 10 then begin
         Left := 10 + (i — 1)*(ImageList1.Width + 7);
         Top := 10;
      end;
      if i > 10 then begin
         Left := 10 + (i — 11)*(ImageList1.Width + 7);
         Top := 17 + ImageList1.Height;
      end;
      Canvas.Brush.Color := Panel1.Brush.Color;
      Canvas.FillRect(Canvas.ClipRect);
      ImageList1.Draw(Canvas, 0, 0, i — 1);
    end;
end;

procedure TfmStartFile.btnCloseClick(Sender: TObject);
begin
  Close;
end;
end.
```

При нажатии кнопки **Каталог** появляется диалоговое окно, где пользователь может указать исходный каталог, в котором выбирается список программ для формирования содержимого панели. Первоначально в качестве исходного предлагается текущий каталог.

Из каталога, указанного пользователем, выбираются первые двадцать исполняемых файлов, имена которых запоминаются в массиве `FileList`, а их значки — в компоненте `ImageList1`. Если у файла отсутствует значок, что характерно, например, для DOS-программ, то он берется из файла dos.ico. Поиск файлов в каталоге выполняется с помощью процедур `FindFirst`, `FindNext` и `FindClose`. Результаты поиска запоминаются в специальной переменной-записи `SR` типа `TSearchRec`. Если поиск был успешным, имя найденного файла находится в поле `SR.Name`. (Подробно вопросы, связанные с файловыми операциями, рассматриваются в *главе 12*.)

Если найден очередной файл, вызывается процедура `AddImage`, в которой запоминается имя файла и извлекается его значок. Последнее выполняется с помощью двух API-функций: `ImageList_AddIcon` и `ExtractIcon`.

Функция `ExtractIcon(hInst: HInstance, lpszExeFileName: LPCTSTR, nIconIndex: UINT): HIcon` *возвращает дескриптор* значка файла, заданного параметром `lpszExeFileName`. Файл может быть исполняемым файлом (exe), динамически загружаемой библиотекой

(dll) или файлом значка (ico). Параметр `nIconIndex` указывает номер значка в файле, если их несколько. Параметр `hInst` идентифицирует вызвавшее эту функцию приложение. Если значок в файле не найден, то функция `ExtractIcon` возвращает значение `0`.

Функция `ImageList_AddIcon(himl: HImageList, hicon: HIcon): Int` *добавляет* к списку изображений, указанному ссылкой `himl`, значок, дескриптор которого задан параметром `hicon`. Если функция выполнена успешно, то в качестве результата возвращается номер нового элемента, добавленного к списку изображений. Если функция не выполнена, то ее результатом является значение `-1`.

После формирования списка имен файлов и их значков динамически создается нужное число компонентов `Image`, на которых отображаются соответствующие значки. Для вывода подсказок свойству `Hint` каждого компонента `Image` в качестве значения присваивается строка с именем программы. Для размещения на панели свойствам `Parent` всех компонентов `Image` задается значение `Panel1`, а реакцию на нажатие этих компонентов обеспечивает установленный для них общий обработчик события `OnClick` — процедура `FileStart`.

В процедуре `FileStart` осуществляется запуск программы, на значке которой был сделан щелчок. Имя запускаемой программы определяется значением свойства `Hint`. В более сложном случае (при одновременной работе с несколькими приложениями) требуется выбрать имя в массиве `FileList`, т. к. при смене другим приложением текущего каталога запускаемые программы могут быть не найдены на диске. При этом в массиве `FileList` должны запоминаться полные имена файлов, т. е. включающие букву устройства и путь.

Для *запуска* из приложения другого приложения использована API-функция `WinExec(lpCmdLine: LPCSTR, uCmdShow: UINT): UINT`, которая выполняет команду, указанную параметром `lpCmdLine`. В командной строке задается имя запускаемого файла, а также дополнительная информация (параметры программы).

С помощью параметра `uCmdShow` выбирается способ отображения окна запускаемой программы. Наиболее часто используются следующие значения этого параметра:

- `SW_ShowMaximized` (новое окно развернуто);
- `SW_ShowMinimized` (новое окно свернуто);
- `SW_ShowNormal` (новое окно отображается в обычном виде).

После запуска новое приложение функционирует в обычном режиме, и его выполнение не зависит от работы вызвавшего его приложения.

Кроме использованной нами API-функции `WinExec`, в Delphi для запуска приложений имеются и другие средства. В частности, к ним относится 32-разрядная функция `ShellExecute`, полностью совместимая с версиями Windows 95 и выше. Однако ее практическое применение ограниченно, поскольку требует программной установки многих параметров. Более удобна функция `ExecuteFile`, входящая в состав модуля `FMXUtils`. Выполняясь, она вызывает функцию `ShellExecute`, но имеет меньше параметров, поэтому использовать ее проще.

Рассмотренную программу можно доработать и реализовать режим, когда пользователь выбирает файлы в требуемых каталогах по одному, задавая в качестве значения

свойства `Hint` произвольный текст, т. е. действуя так же, как при создании ярлыков приложений в Windows. После завершения программы конфигурация панели инструментов и информация о связанных с ней файлах должна сохраняться в инициализационном файле (или системном реестре) для восстановления при последующем запуске. Именно так работает панель инструментов Microsoft Office.

Для отображения значка и запуска файлов вместо компонента `Image` можно использовать другие элементы управления, например быстрые кнопки `SpeedButton`. Размещать эти элементы можно не только на компоненте `Panel`, как это сделано в примере, но и на панели инструментов `ToolBar` или в строке состояния `StatusBar`.

Создание панели инструментов на основе компонентов *ToolBar* и *CoolBar*

На практике для создания панелей инструментов удобно использовать специальные компоненты `ToolBar` и `CoolBar`, находящиеся на странице **Win32** Палитры компонентов.

Компонент *ToolBar*

Компонент `ToolBar` содержит специальные (инструментальные) кнопки `ToolButton` и позволяет манипулировать ими. Реализуемая этим компонентом панель инструментов автоматически задает для всех своих кнопок одинаковые размеры, упорядочивает их расположение и при необходимости переносит на другую строку. Управлять панелью инструментов и ее отдельными элементами можно на этапе конструирования или динамически во время выполнения приложения. Управление панелью заключается в добавлении или уничтожении отдельных элементов, а также в изменении свойств этих элементов. Для этих целей разработчик использует свойства компонента `ToolBar` и находящихся на нем компонентов.

Кроме кнопок `ToolButton`, на панель `ToolBar` можно помещать и другие компоненты управления, например, быстрые кнопки `SpeedButton`, комбинированные списки `ComboBox` или однострочные редакторы `Edit`. Эти компоненты также размещаются и упорядочиваются автоматически.

При разработке приложения для добавления к панели инструментов новой кнопки `ToolButton` нужно щелкнуть на компоненте `ToolBar` правой кнопкой мыши и выбрать пункт **New Button** (Новая кнопка) контекстного меню, в результате чего справа от имеющихся кнопок `ToolButton` появится новая кнопка. Инструментальные кнопки автоматически выравниваются и размещаются вплотную друг к другу. Чтобы визуально выделить отдельную кнопку или их группу, в панель инструментов можно вставить разделитель. Для этого в контекстном меню следует выбрать команду **New Separator** (Новый разделитель).

В пределах панели отдельные элементы можно перемещать с помощью мыши. По окончании перемещения все элементы панели автоматически выравниваются. В случае изменения размеров какого-либо элемента, например высоты (свойство `Height`), остальные элементы также изменят свой размер.

После помещения кнопки `ToolButton` на панель инструментов ее поведением можно управлять через Инспектор объектов. Кнопка `ToolButton` является потомком класса

TGraphicControl и во многом схожа с кнопкой быстрого доступа SpeedButton. Работа с отдельными инструментальными кнопками и их группами в процессе создания панели инструментов почти не отличается от рассмотренных ранее действий с кнопками быстрого доступа, однако имеет свои особенности.

Вид кнопки ToolButton определяется свойством Style типа TToolButtonStyle, которое может принимать следующие значения:

- tbsButton (обычная кнопка);
- tbsCheck (переключатель);
- tbsDropDown (стрелка вниз, при нажатии раскрывающая меню); меню типа TPopupMenu должно быть предварительно подготовлено, а ссылка на конкретное меню задается значением свойства DropdownMenu типа TPopupMenu кнопки;
- tbsSeparator (разделитель в виде пустого промежутка) — для визуального отделения элементов и разграничения отдельных групп инструментальных кнопок;
- tbsDivider (разделитель в виде вертикальной линии) — выполняет те же функции, что и разделитель tbsSeparator.

Для оформления панели инструментов в *современном стиле* интерфейса Windows (с плоскими кнопками) нужно установить свойство Flat (типа Boolean) в значение True. Если панель инструментов ToolBar расположена на панели инструментов CoolBar, то свойство Transparent (типа Boolean) кнопок также устанавливается в True. В этом случае кнопки будут прозрачными для общего фона панели инструментов.

Для *объединения* кнопок ToolButton в группу используется их свойство Grouped типа Boolean. По умолчанию оно имеет значение False, и все кнопки работают независимо друг от друга. Чтобы образовать группу взаимосвязанных кнопок, когда выбор любой из них делает другие кнопки невыбранными, необходимо установить свойства Grouped всех объединяемых кнопок панели в значение True. Кроме того, кнопки группы должны иметь стиль переключателя tbsCheck и отделяться от других групп и отдельных кнопок разделителем tbsSeparator или tbsDivider. В качестве разделителей групп могут выступать и некнопочные элементы, например компоненты ComboBox и Edit.

При создании независимого переключателя компонент ToolButton также имеет стиль tbsCheck, а его свойство AllowAllUp, как и в случае с быстрой кнопкой SpeedButton, устанавливается в значение True. При этом значение свойства Grouped переключателя не важно.

Если кнопка ToolButton дублирует команду меню, то ее свойство MenuItem типа TMenuItem может указывать на соответствующий пункт меню. Если свойство MenuItem указывает на пункт меню, то при нажатии кнопки команда, закрепленная за этим пунктом, будет выполнена без написания обработчика события OnClick кнопки. Однако надо учитывать, что при установке этого свойства как указателя пункта меню одновременно изменяются свойства Caption и ImageIndex кнопки. Значение свойства Caption кнопки становится равным значению свойства Caption ассоциированного пункта меню, а свойство ImageIndex принимает значение –1, т. е. на поверхности кнопки ToolButton не отображается рисунок. Если такие изменения не нужны, то после присвоения значения свойству MenuItem следует изменить заголовок и ссылку на рисунок кнопки ToolButton (восстановить значения этих свойств).

Вывод изображения на поверхности кнопки `ToolButton` имеет некоторые особенности по сравнению с ранее рассмотренными кнопками `BitBtn` и `SpeedButton`. Рисунок для кнопки `ToolButton` определяется свойством `ImageIndex` типа `Integer`, значение которого является целым числом, определяющим порядковый номер рисунка в списке (отсчет номеров начинается с нуля). Если задан номер несуществующего рисунка, то изображение на поверхности кнопки отсутствует. Если отображать рисунок не нужно, то можно задать свойству `ImageIndex` значение `-1`. Список рисунков обычно находится в компоненте `ImageList`, содержимое которого формируется при конструировании приложения.

Отображаемые на кнопках рисунки представляют собой изображения типа BMP, как правило, размером 16×16 пикселов, и могут быть подготовлены с помощью любого графического редактора.

Для кнопок `ToolButton` в качестве основных выбираются рисунки, содержащиеся в списке, заданном свойством `Images` панели инструментов.

Если кнопка используется как переключатель, то она может отображать второй рисунок, соответствующий ее *включенному* (выбранному) состоянию. Второй рисунок выбирается из другого списка `ImageList`, который также формируется заранее и указывается свойством `HotImages` панели инструментов.

Неактивные, или *заблокированные*, кнопки отображают рисунок из третьего списка, указываемого свойством `DisableImages` панели инструментов. Если этот рисунок отсутствует, то выводится основной рисунок, но более бледным цветом.

При изменении размеров формы, а вместе с ней и панели инструментов, некоторые кнопки `ToolButton` перестают умещаться в отведенной области и могут быть *автоматически перенесены* на следующую строку. Этой возможностью управляет свойство `Wrap` типа `Boolean`: если оно имеет значение `True`, то автоматический перенос кнопок разрешен; при этом значение `True` должно быть установлено также для свойства `Wrapable` типа `Boolean` самой панели инструментов. По умолчанию свойство `Wrap` имеет значение `False`, и автоматический перенос кнопок не выполняется.

Кнопки `ToolButton` могут быть *маркированы*, а также находиться в третьем, *неопределенном*, состоянии. Этими состояниями управляют свойства `Marked` и `Indeterminate` типа `Boolean`, по умолчанию имеющие значения `False`. При установке какого-либо из названных свойств в значение `True` кнопка отображает это соответствующим визуальным эффектом. В программе можно предусмотреть анализ свойств `Marked` и `Indeterminate` кнопок панели инструментов и в зависимости от их значений выполнить определенные действия.

Свойства компонента `ToolBar` определяют его поведение, а также поведение содержащихся в нем элементов. Далее кратко описаны наиболее важные из этих свойств.

По умолчанию на кнопках `ToolButton` их *заголовки* (текст) не отображаются, даже если значение их свойств `Caption` отлично от пустой строки. Это обусловлено свойством `ShowCaption` типа `Boolean` панели инструментов, которое по умолчанию имеет значение `False`. Если это свойство установить в значение `True`, то на поверхности кнопок могут отображаться одновременно рисунок и текст. Даже если текст не отображается, кнопка

все же реагирует на комбинации быстрых клавиш, заданных для нее в свойстве Caption.

Свойство Flat типа Boolean управляет *видом* кнопок ToolButton панели инструментов. Значение этого свойства, равное False (по умолчанию), соответствует обычному объемному виду кнопок, а установка свойству значения True задает плоские кнопки современного стиля.

Свойство Indent типа Integer задает для панели инструментов *отступ слева* в пикселах (свободный от элементов участок), по умолчанию имеет значение 0.

Если на поверхности кнопок ToolButton одновременно отображаются и заголовок, и рисунок, то свойство List типа Boolean управляет их *взаимным расположением*. По умолчанию оно имеет значение False, и заголовок находится под рисунком. Если свойство List имеет значение True, то рисунок появляется слева от заголовка.

Свойство RowCount типа Integer, доступное только для чтения и только во время выполнения программы, содержит *число строк*, занимаемых панелью инструментов. Значение этого свойства может быть проанализировано, например, в обработчике события OnResize панели инструментов.

Напомним, что свойство Images типа TCustomImageList задает список ImageList, содержащий рисунки для всех кнопок ToolButton панели инструментов. Аналогично свойство HotImages типа TCustomImageList указывает список с рисунками для *выбранного* состояния кнопок-переключателей. Свойство DisableImages типа TCustomImageList позволяет указать список для третьего, *невыбранного* состояния кнопки. Если какой-либо из списков не задан, то вместо рисунка, соответствующего этому состоянию, отображается основной рисунок из первого списка. Если отсутствует и этот рисунок, то на поверхности кнопки отображается только заголовок.

Ряд свойств позволяет управлять *внешним видом* панели инструментов.

Свойство EdgeBorders типа TEdgeBorders определяет наличие или отсутствие *разделительной рамки* с каждой из четырех сторон панели инструментов. Это свойство может принимать комбинации следующих значений:

- ebLeft (линия слева);
- ebTop (линия сверху);
- ebRight (линия справа);
- ebBottom (линия снизу).

По умолчанию свойство EdgeBorders имеет значение ebTop.

Свойства EdgeInner и EdgelOuter типа TEdgeStyle задают вид внешней и внутренней *фасок* и принимают следующие значения:

- esNone (нет фаски);
- esRaised (приподнятая фаска);
- esLowered (углубленная фаска).

По умолчанию внешняя фаска выглядит углубленной, а внутренняя — приподнятой относительно поверхности формы.

Свойство `BorderWidth` типа `TBorderWidth` задает *ширину границы* (бордюра) панели инструментов (в пикселах), по умолчанию имеет значение `0`.

Управлять панелью инструментов можно и динамически, т. е. при выполнении приложения. При этом можно не только изменить поведение отдельных элементов панели, но и добавить или удалить их. Так, в следующем примере к панели инструментов `ToolBar1` добавляется кнопка `tbtNew`:

```
var tbtNew: TToolButton;
...
tbtNew := TToolButton.Create(Self);
tbtNew.Parent := ToolBar1;
```

Для доступа к кнопкам `ToolButton` во время работы программы удобно использовать свойство `Buttons[Index: Integer]` типа `TToolButton`. При этом первая кнопка обозначается `Buttons[0]`, вторая — `Buttons[1]` и т. д. Возможно обращение к кнопкам и по именам. Например, результаты работы двух следующих строк

```
ToolBar2.Buttons[2].Enabled := False;
tbtnSave.Enabled := False;
```

одинаковы — блокирование третьей кнопки, имеющей имя `tbtnSave` и принадлежащей панели `ToolBar2`.

Рассмотрим пример создания в форме панели инструментов, вид которой при конструировании приложения показан на рис. 5.13. Панель инструментов представляет собой компонент `ToolBar1`, на котором расположено десять кнопок `ToolButton` и редактор `Edit`. Кроме того, в форме находятся главное меню `MainMen1`, локальное меню `MenuPopup1`, а также три компонента `ImageList`.

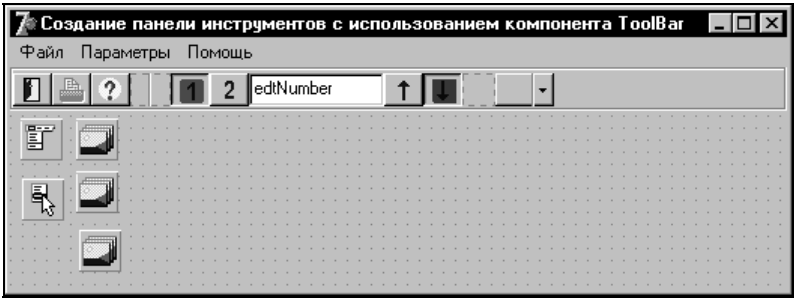


Рис. 5.13. Вид панели инструментов при конструировании приложения

Первые три кнопки панели инструментов дублируют соответствующие команды главного меню `MainMen1`; их имена (слева направо) `tbtnExit`, `tbtnPrint` и `tbtnHelp`. Кнопка `tbtnPrint` заблокирована, т. к. операция печати сразу после запуска приложения невозможна. Далее расположен разделитель в виде вертикальной линии `tbtnSeparator1`, за которым находится группа зависимых переключателей `tbtnShow1` и `tbtnShow2`. Переключатель `tbtnShow1` первоначально находится в выбранном состоянии.

Редактор `edtNumber` предназначен для ввода кодового номера, одновременно он отделяет предыдущую группу переключателей от двух независимых переключателей, распо-

ложенных справа: tbtnEffect1 и tbtnEffect2, первоначально выбран второй из них. Последней в панели инструментов располагается кнопка tbtnMenu, которая при нажатии вызывает локальное меню MenuPopup1. От предыдущих элементов кнопку вызова меню отделяет разделитель — кнопка tbtnSeparator2.

Списки ImageList содержат рисунки, отображаемые на кнопках панели инструментов.

Главное и локальное меню подготовлены с помощью Конструктора меню, на рисунке видна строка главного меню.

В листинге 5.4 приводятся фрагменты модуля uExtend5 с описанием формы Form5, относящиеся к созданию панели инструментов на основе компонента ToolBar1.

Листинг 5.4. Пример панели инструментов на основе компонента ToolBar1

```
unit uExtend5;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ToolWin, ComCtrls, Menus, StdCtrls, Buttons, ImgList;
type
  TForm5 = class(TForm)
      MainMenu1: TMainMenu;
        mnuFile: TMenuItem;
       mnuPrint: TMenuItem;
        mnuExit: TMenuItem;
    mnuOptions: TMenuItem;
       mnuHelp: TMenuItem;
    ImageList1: TImageList;
    ImageList2: TImageList;
      ToolBar1: TToolBar;
      tbtnExit: TToolButton;
     tbtnPrint: TToolButton;
      tbtnHelp: TToolButton;
 tbtnSeparator1: TToolButton;
     tbtnShow1: TToolButton;
     tbtnShow2: TToolButton;
   tbtnEffect1: TToolButton;
   tbtnEffect2: TToolButton;
     edtNumber: TEdit;
    ImageList3: TImageList;
    PopupMenu1: TPopupMenu;
      tbtnMenu: TToolButton;
 tbtnSeparator2: TToolButton;
...
    procedure FormCreate(Sender: TObject);
...
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form5: TForm5;
implementation
{$R *.DFM}

// Удобнее выполнить эти действия при разработке
// приложения с помощью Инспектора объектов
procedure TForm5.FormCreate(Sender: TObject);
begin
  mnuPrint.Enabled := False;
  // Установка свойств панели инструментов
  ToolBar1.EdgeBorders := [ebLeft,ebTop,ebRight,ebBottom];
  ToolBar1.Images := ImageList1;
  ToolBar1.HotImages := ImageList2;
  ToolBar1.DisabledImages := ImageList3;
  ToolBar1.Flat := False;
  ToolBar1.ShowCaptions := False;
  ToolBar1.List := True;
  ToolBar1.Wrapable := False;
  // Установка свойств кнопок
  tbtnExit.Style := tbsButton;
  tbtnExit.MenuItem := mnuExit;
  tbtnExit.Caption := '';
  tbtnExit.ImageIndex := 0;
  tbtnPrint.Style := tbsButton;
  tbtnPrint.MenuItem := mnuPrint;
  tbtnPrint.Caption := '';
  tbtnPrint.ImageIndex := 1;
  tbtnPrint.Enabled := False;
  tbtnHelp.Style := tbsButton;
  tbtnHelp.MenuItem := mnuHelp;
  tbtnHelp.Caption := '';
  tbtnHelp.ImageIndex := 2;
  // Установка свойств группы из двух зависимых переключателей
  tbtnSeparator1.Style := tbsDivider;
  tbtnSeparator1.Width := 50;
  tbtnShow1.Style := tbsCheck;
  tbtnShow1.Caption := '';
  tbtnShow1.ImageIndex := 3;
  tbtnShow1.Grouped := True;
  tbtnShow1.AllowAllUp := True;
  tbtnShow2.Style := tbsCheck;
  tbtnShow2.Caption := '';
  tbtnShow2.ImageIndex := 4;
  tbtnShow2.Grouped := True;
  tbtnShow2.AllowAllUp := True;
  tbtnShow1.Down := True;
  // Установка свойств независимых переключателей
  tbtnSeparator2.Style := tbsSeparator;
  tbtnSeparator2.Width := 30;
  tbtnEffect1.Style := tbsCheck;
```

```
  tbtnEffect1.Caption := '';
  tbtnEffect1.ImageIndex := 5;
  tbtnEffect1.Grouped := False;
  tbtnEffect1.AllowAllUp := True;
  tbtnEffect2.Style := tbsCheck;
  tbtnEffect2.Caption := '';
  tbtnEffect2.ImageIndex := 6;
  tbtnEffect2.Grouped := False;
  tbtnEffect2.AllowAllUp := True;
  tbtnEffect2.Down := True;
  // Задание значения редактора
  edtNumber.Text := '137';
  // Установка свойств кнопки вызова меню
  tbtnMenu.Style := tbsDropDown;
  tbtnMenu.DropDownMenu := PopupMenu1;
end;
...
```

Установка значений свойств панели инструментов и ее отдельных элементов происходит в процессе выполнения приложения при создании формы `Form5`. На практике эти действия удобно выполнять с помощью Инспектора объектов на этапе конструирования приложения.

Компонент *CoolBar*

Панель инструментов, реализуемая компонентом `CoolBar`, содержит несколько специальных полос, определяемых свойством `Bands` типа `TCoolBands`. Эти полосы могут содержать другие элементы управления (в том числе компоненты `ToolBar`), которые способны перемещаться и изменять свои размеры. Примером использования панели инструментов `CoolBar` может служить панель кнопок в программном продукте Microsoft Internet Explorer 5.0.

Создание панели инструментов на основе компонента *Form*

Форму как компонент-контейнер также можно использовать для создания панели инструментов. Для этого в нее помещаются нужные интерфейсные элементы, а свойство `BorderStyle` устанавливается в значение `bsSizeToolWin` или `bsToolWindow`. Тогда область заголовка формы принимает вид, как у панели инструментов, — отсутствует кнопка вызова системного меню и независимо от значения свойства `BorderIcons` присутствует кнопка закрытия окна (рис. 5.14). Форма-панель инструментов имеет заголовок и ее можно перемещать мышью, т. е. такая панель является *плавающей*.

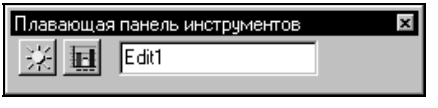


Рис. 5.14. Форма-панель инструментов

Если свойство `BorderStyle` имеет значение `bsSizeToolWin`, то размеры панели инструментов можно изменять с помощью мыши. В случае, когда свойство `BorderStyle` имеет значение `bsToolWindow`, размеры панели инструментов визуально изменять нельзя. Однако на программном уровне управление размерами формы-панели инструментов реализуется в любом случае с помощью соответствующего кода. Кроме того, поскольку такая панель является окном, его можно перемещать с помощью мыши, как любое другое окно. При этом, в отличие от рассмотренных ранее видов панелей инструментов, такое перемещение не требует специального кодирования и выполняется автоматически.

При разработке многодокументного приложения панель инструментов является дочерним окном и может перемещаться в пределах главного (родительского) окна. В случае создания однодокументного приложения форма-панель инструментов независима от других окон, и пользователь имеет право разместить ее в любом месте экрана.

Механизм действий

Характеристика механизма действий

Механизм действий предназначен для обеспечения возможности удобного задания централизованной реакции приложения на действия пользователя. Поскольку определенные действия пользователя могут быть заданы с помощью нескольких различных интерфейсных элементов (пунктов меню, кнопок панелей инструментов и др.), то этим и обуславливается необходимость использования единой централизованной реакции на них.

При описании механизма действий будем использовать следующий ряд основных понятий. *Действие* представляет собой ответную реакцию приложения на некоторые действия пользователя, такие как выбор пункта меню или нажатие кнопки панели инструментов. *Клиент действия*, как правило, это пункт меню или кнопка панели инструментов, с помощью которых выполняется активизация действия. *Цель действия* представляет собой тот компонент (элемент управления) на форме, над которым выполняются операции действия, например, это может быть редактор, с которым ведется обмен информацией с буфером обмена.

Для поддержки механизма действий в Delphi имеется невизуальный компонент `ActionList` типа `TActionList`, который размещен на странице **Standard** Палитры компонентов. Кроме того, класс `TComponent` имеет свойство `Action`, с помощью которого для элемента управления может быть назначено требуемое действие.

В компоненте `ActionList` хранится список имен действий и имен соответствующих им подпрограмм, реализующих эти действия. Использование этого компонента выполняется следующим образом. Сначала на форму помещается компонент `ActionList` типа `TActionList`, затем двойным щелчком мыши на компоненте или с помощью команды **Action List Editor** контекстного меню компонента вызывается редактор действий (рис. 5.15).

В редакторе действий можно выполнить добавление пользовательского действия или стандартного действия. При добавлении *пользовательские действия* по умолчанию получают имена **Action1**, **Action2**, ...

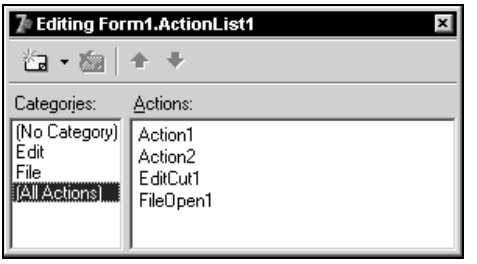


Рис. 5.15. Окно редактора действий

Стандартные действия сгруппированы по функциональному назначению, их добавление выполняется с помощью диалогового окна **Standard Action Classes** (рис. 5.16), вызываемого с помощью команды **New Standard Action** контекстного меню Редактора действий.

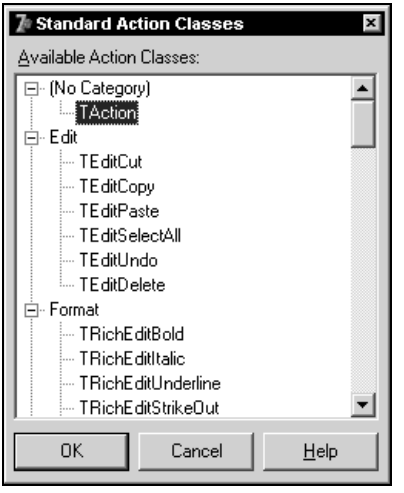


Рис. 5.16. Диалоговое окно **Standard Action Classes**

Имя определенного действия присваивается свойству `Action` тех элементов управления (команд меню или кнопок панелей инструментов), использование которых должно сопровождаться именно этим действием.

Связанная с действием (пользовательским или стандартным) группа свойств (`Caption`, `Checked`, `Enabled` и др.) присваивается одноименным свойствам компонентов, для которых это действие устанавливается. На кнопках панели инструментов отображаются рисунки (если заданы) и названия действий, а для команд меню, кроме того, отображаются сочетания клавиш (если установлены) (рис. 5.17).

Если в качестве действия для ряда элементов управления (команд меню или кнопок панелей инструментов) устанавливается *стандартное* действие, то при активизации любого из них это действие выполняется автоматически. Делается это при помощи соответствующих свойств компонента действия, которые можно просмотреть с помощью Инспектора объектов. Например, для стандартного компонента действия `FilePrintSetup1`

реализуемое им действие определяется его свойством `Dialog`, которое имеет значение `PrinterSetupDialog`.

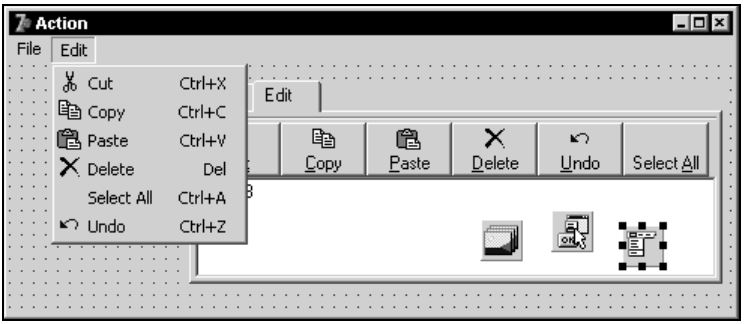


Рис. 5.17. Вид меню и панели инструментов, использующих общие действия

Если в качестве действия для элементов управления устанавливается *пользовательское* действие, то для этого действия обязательно нужно задать обработчик события `OnExecute`. Например, если мы хотим, чтобы при реализации пользовательского действия выполнялся вызов диалога открытия файла, то нужно задать следующий обработчик события `OnExecute`:

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
   OpenDialog1.Execute;
end;
```

При этом на форме приложения нужно поместить компонент `OpenDialog` типа `TOpenDialog`, который размещается на странице **Dialogs** Палитры компонентов. Если же для тех же целей использовать стандартный компонент действия `FileOpen`, то компонент `OpenDialog` на форме размещать не требуется.

Стандартные действия

Как отмечалось, стандартные действия сгруппированы по функциональному назначению (см. рис. 5.1). А именно, стандартные действия разделены на ряд категорий, в том числе такие, как:

- **Edit** — редактирования, включает операции: вырезания, копирования, вставки, выделения всего содержимого, отмены и удаления;
- **Format** — форматирования для элементов RichEdit, включает операции выбора начертания: полужирного, курсива, подчеркивания, назначения маркеров абзацев и различных вариантов выравнивания;
- **Help** — справки, служат для вызова различных страниц справочной помощи файла справки, присоединенного к приложению;
- **Window** — управления окнами приложения, включает операции с окнами: закрытия, каскадного, горизонтального и вертикального размещения окон, сворачивания окон, упорядочения окон;

- **File** — работы с файлами, включая операции вызова диалогов: открытия, сохранения, настройки печати, настройки страниц, запуска и выхода;
- **Search** — поиска, включает операции: поиска, поиска следующего вхождения, замены и поиска первого вхождения;
- **Tab** — управления страницами, включает операции перехода к следующей и предыдущей страницам;
- **List** — действия со списками, включает операции копирования, удаления, выделения всех строк, очистки и перемещения. Кроме того, включает операции определения статических и виртуальных списков, связываемых с пользовательским интерфейсом;
- **Dialog** — управления диалоговыми окнами, включает операции: вызова диалоговых окон для открытия и сохранения файлов рисунков, для выбора цвета, шрифта, для печати;
- **Internet** — работы с Интернетом, включает операции: просмотра по URL-адресу, загрузки файла по URL-адресу и отправки почты;
- **Dataset** — действия с наборами данных, касаются работы с таблицами и запросам баз данных;
- **Tools** — настройка панели действий в диалоговом окне.

При использовании ряда стандартных действий, например, действий из категории редактирования, требуется определить цель действия, т. е. компонент, в отношении которого будет выполняться действие.

При определении цели действия может использоваться метод `HandlesTarget` этого действия. Он проверяет, применимо ли действие к цели `Target`. В случае положительного ответа осуществляется выполнение действия путем вызова его метода `ExecuteTarget`.

Определение цели действия выполняется следующим образом: прежде всего, выбирается активный элемент управления на форме, если такового нет или он не подходит (о чем свидетельствует метод `HandlesTarget`), то в качестве цели рассматривается текущая форма. Если текущая форма также не подходит, то перебираются все компоненты на форме.

Менеджер действий

Менеджер действий представлен компонентом `ActionMenuBar` типа `TActionMenuBar`, расположенным на странице **Additional** Палитры компонентов. В сравнении с компонентом `ActionList` типа `TActionList` он предоставляет некоторые дополнительные возможности по созданию действий в привязке к главному меню и специальным панелям инструментов (компонентам типа `TActionToolBar`), конструируемых визуально с помощью компонента `ActionMenuBar`.

Работа по настройке действий с помощью компонента `ActionMenuBar` ведется в окне Редактора менеджера действий (рис. 5.18), вызов которого выполняется двойным щелчком или командой **Customize** (Настройка) контекстного меню компонента.

При открытой вкладке **Toolbars** (Панели инструментов) Редактора менеджера действий с помощью кнопок **New** и **Delete** можно выполнить добавление и удаление одной или нескольких панелей инструментов (компонентов типа `TActionToolBar`) на форме приложения.

При открытой вкладке **Actions** (Действия) (рис. 5.18) с помощью команд контекстного меню Редактора менеджера действий можно выполнить добавление пользовательских и стандартных действий в поле **Actions** (Действия) вкладки. После этого можно разместить любое из добавленных действий на любую из созданных Панелей инструментов путем перетаскивания его методом drag-and-drop.

Чтобы назначить любое из добавленных действий команде главного меню, достаточно с помощью Инспектора объектов установить свойству `Action` этой команды в качестве значения имя нужного действия.

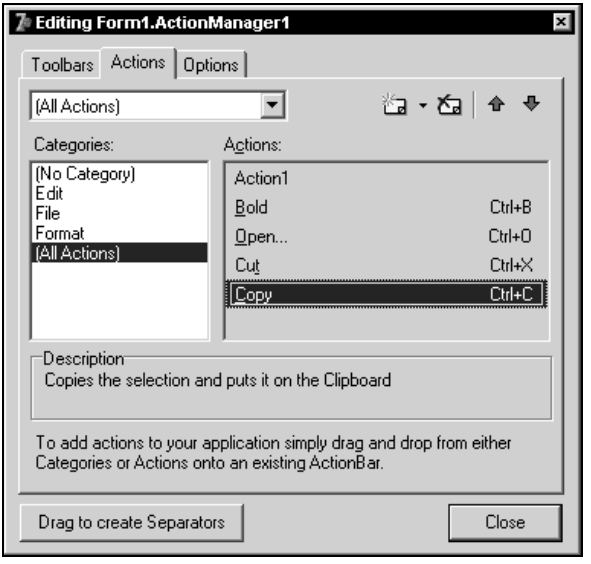


Рис. 5.18. Окно редактора менеджера действий

Таким образом, использование механизма действий позволяет облегчить работу по заданию обработчиков событий выбора пунктов меню или нажатия кнопок панелей инструментов. Наибольшее удобство, как мы видели из приведенных примеров, достигается при установке стандартных действий.

Пример синхронизации элементов управления

Рассмотрим пример синхронизации элементов управления при создании простейшего текстового редактора. Вид соответствующей формы при разработке (слева) и при выполнении (справа) показан на рис. 5.19.

В листинге 5.5 приведен код модуля `uActionList` формы приложения-редактора.

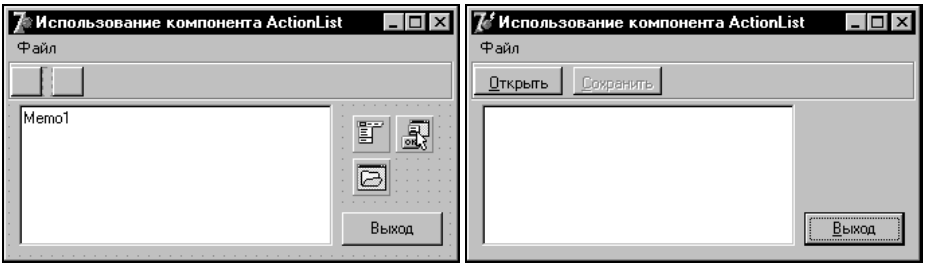


Рис. 5.19. Виды формы приложения-редактора

Листинг 5.5. Пример синхронизации элементов управления редактора

```
unit uActionList;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Menus, ComCtrls, ToolWin, ActnList, ImgList;
type
  TForm1 = class(TForm)
      MainMenu1: TMainMenu;
        mnuFile: TMenuItem;
        mnuOpen: TMenuItem;
        mnuSave: TMenuItem;
        mnuExit: TMenuItem;
       ToolBar1: TToolBar;
       tbtnOpen: TToolButton;
    ToolButton2: TToolButton;
       tbtnSave: TToolButton;
        btnExit: TButton;
          Memo1: TMemo;
    OpenDialog1: TOpenDialog;
    ActionList1: TActionList;
     ActionOpen: TAction;
     ActionSave: TAction;
     ActionExit: TAction;

    procedure FormCreate(Sender: TObject);
    procedure ActionOpenExecute(Sender: TObject);
    procedure ActionSaveExecute(Sender: TObject);
    procedure ActionExitExecute(Sender: TObject);
    procedure ActionSaveUpdate(Sender: TObject);
    procedure ActionExitUpdate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

// Задание начальных свойств для компонентов формы
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Задание способа отображения рисунка и текста
  // на кнопках панели инструментов
  ToolBar1.Images := nil;
  ToolBar1.ShowCaptions := True;
  // Связь пунктов меню и кнопок с объектами Action
  mnuOpen.Action  := ActionOpen;
  tbtnOpen.Action := ActionOpen;
  mnuSave.Action  := ActionSave;
  tbtnSave.Action := ActionSave;
  mnuExit.Action  := ActionExit;
  btnExit.Action  := ActionExit;
  // Очистка многострочного редактора Memo1
  Memo1.Clear;
  Memo1.Modified := False;
end;

// Открытие файла в диалоговом окне OpenDialog1
procedure TForm1.ActionOpenExecute(Sender: TObject);
begin
  if OpenDialog1.Execute then
    Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
end;

// Сохранение редактируемого файла
procedure TForm1.ActionSaveExecute(Sender: TObject);
begin
  Memo1.Lines.SaveToFile(OpenDialog1.FileName);
  Memo1.Modified := False;
end;

// Закрытие формы и прекращение работы приложения
procedure TForm1.ActionExitExecute(Sender: TObject);
begin
  Close;
end;

// Блокировка/разблокировка пункта меню и кнопки панели
// инструментов, связанных с сохранением редактируемого файла
procedure TForm1.ActionSaveUpdate(Sender: TObject);
begin
  if not Memo1.Modified then begin
```

```
    // Блокировка пункта меню и кнопки, если файл не изменялся
    mnuSave.Enabled := False;
    tbtnSave.Enabled := False;
  end
  else begin
    // Разблокировка пункта меню и кнопки, если файл изменен
    mnuSave.Enabled := True;
    tbtnSave.Enabled := True;
  end;
end;

// Блокировка и разблокировка элементов управления,
// связанных с открытием файла и закрытием формы
procedure TForm1.ActionExitUpdate(Sender: TObject);
begin
  if Memo1.Modified then begin
    // Блокировка элементов, если файл изменен
    mnuExit.Enabled  := False;
    btnExit.Enabled  := False;
    mnuOpen.Enabled  := False;
    tbtnOpen.Enabled := False;
  end
  else begin
    // Разблокировка элементов, если файл не изменялся или был сохранен
    mnuExit.Enabled  := True;
    btnExit.Enabled  := True;
    mnuOpen.Enabled  := True;
    tbtnOpen.Enabled := True;
  end
end;

end.
```

Отображение и редактирование текста выполняется с помощью компонента `Memo1`. Выбор имени редактируемого файла осуществляется в стандартном диалоге `OpenDialog1`. Форма содержит три вида элементов управления: главное меню `MainMenu1`, панель инструментов `ToolBar1` и кнопку `btnExit`.

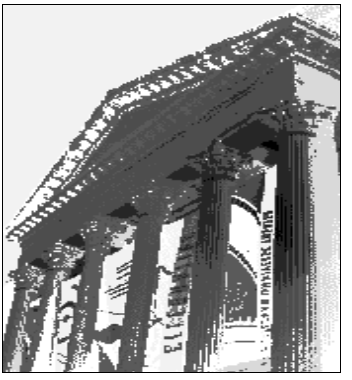
Меню содержит подменю **Файл** (`mnuFile`), состоящее из трех пунктов: **Открыть** (`mnuOpen`), **Сохранить** (`mnuSave`) и **Выход** (`mnuExit`). Первые два пункта дублируются кнопками панели инструментов. Последний пункт меню дублируется кнопкой **Выход** (`btnExit`). В скобках указаны значения свойства `Name` элементов управления. Заголовки пунктов меню установлены через Инспектор объектов.

Для синхронизации указанных элементов используется компонент `ActionList1`. При разработке приложения с помощью специального редактора к нему добавлены три объекта-действия `ActionOpen`, `ActionSave` и `ActionExit`, которые соответственно предназначены для централизованного управления операциями открытия файла, сохранения файла и выхода из приложения. Этим объектам присвоены имена `&Открыть`, `&Сохранить` и `&Выход`.

Для каждого из объектов `Action` создан обработчик события `OnExecute`, в котором располагаются инструкции, выполняющие требуемые действия. С целью управления режимом доступа к кнопкам и пунктам меню, связанным с соответствующими действиями, для объектов `Action` (кроме `ActionOpen`) созданы обработчики события `OnUpDate`. При разрешении или запрещении (установкой значения свойства `Enabled`) доступа к элементам управления анализируется значение свойства `Modified` многострочного редактора. Например, если редактируемый текст после изменения не был сохранен, то блокируется кнопка `btnOpen` открытия нового файла.

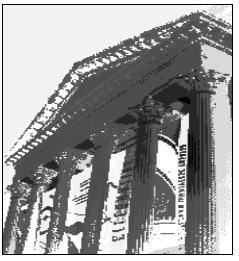
Связь элементов управления с соответствующими объектами-действиями выполнена при создании формы.

После запуска программы в качестве заголовков элементов управления выводятся значения свойства `Caption` объектов-действий, связанных с этими элементами. Для пунктов меню и простой кнопки на время выполнения программы они заменяют их собственные значения свойств `Caption`, заданные через Инспектор объектов при разработке. Кнопки панели инструментов, для которых на этапе разработки не был задан заголовок, получают его от объектов-действий. При выполнении программы кнопки панели инструментов автоматически изменяют размер в зависимости от длины заголовка.



ЧАСТЬ II

Развитые средства Delphi



ГЛАВА 6

Управление приложением и экраном

Часто создание прикладных программ (приложений) требует от разработчика действий по управлению приложением и экраном как отдельными объектами. В Delphi эта возможность обеспечивается объектами `Application` класса `TApplication` и `Screen` класса `TScreen`, которые представляют собой приложение и экранную среду его выполнения соответственно.

Объект *Application*

Приложение в целом описывается классом `TApplication`. При каждом запуске Delphi автоматически создает объект типа `TApplication` с именем `Application`. По своей сути объект `Application` является окном Windows со стандартной оконной процедурой API `WndProc`, однако Delphi скрывает неудобный для программиста способ общения с приложением на уровне функций API и предоставляет возможность работать на более высоком уровне.

Объект `Application` недоступен при конструировании приложения, но его автоматическое создание отображается в исходном файле проекта (dpr):

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

В начале работы над новым приложением к файлу проекта автоматически добавляются вызовы трех методов: `Initialize`, `CreateForm` и `Run`.

Процедура `Initialize` *инициализирует* приложение в целом, при этом также выполняются операторы раздела `initialization` всех модулей приложения.

Процедура `CreateForm(FormClass: TFormClass; var Reference)` *создает* форму, тип которой задается параметром `FormClass`, и присваивает ее переменной, определяемой параметром `Reference`. Используя этот параметр, можно получить доступ к созданной форме и ее компонентам. Владельцем созданной формы является объект `Application`. Процедура `CreateForm` обычно вызывается автоматически для каждой формы в составе приложения. При необходимости можно отменить автоматическое создание формы, исключив эту процедуру из файла проекта. Однако перед любым использованием формы она должна быть уже создана, иначе произойдет ошибка.

Процедура `Run` *запускает* приложение, обычно ее вызов является последним оператором проекта.

Часто использование объекта `Application` ограничивается указанными тремя методами, однако у него есть и другие методы, свойства и события, применение которых позволяет повысить качество разрабатываемых приложений.

Свойство `Title` типа `String` определяет *название* приложения и содержит текст, идентифицирующий приложение. По умолчанию название приложения совпадает с именем exe-файла программы, первоначально это `project1`. При необходимости можно сделать название приложения (именно он отображается в панели задач при свертывании приложения) совпадающим с заголовком главной формы, например, так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.Title := 'Программа тестирования';
  Form1.Caption := Application.Title;
end;
```

Свойство `ExeName` типа `String` содержит *путь* и *имя* исполняемого файла приложения — *спецификацию* exe-файла. Оно доступно для чтения на этапе выполнения приложения.

Пример использования свойства `ExeName`:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := Application.ExeName;
end;
```

При необходимости из спецификации файла с помощью функций `ExtractFileName` и `ExtractFilePath` можно получить путь и имя файла. Например, в следующем примере выделяется название каталога (путь), из которого запущено приложение.

```
var DirAppl: string;
...
// В конце строки, содержащей каталог, имеется обратный слэш (\)
DirAppl := ExtractFilePath(Application.ExeName);
```

Свойство `Active` типа `Boolean` определяет *активность* приложения. Во время выполнения приложения оно доступно только для чтения. Свойство `Active` имеет значение `True`, когда приложение активно и содержит фокус ввода, в этом случае одноименное свойство одной из форм также имеет значение `True`. В противном случае это свойство имеет значение `False`.

Свойство `HelpFile` типа `String` указывает *имя справочного файла*, используемого приложением. Оно доступно во время выполнения приложения. По умолчанию значением свойства `HelpFile` является пустая строка. Если файл не задан, то методы и свойства, относящиеся к справке, не действуют.

Если приложение использует один справочный файл, то его имя обычно указывается в файле проекта или в обработчике события создания главной формы, например, так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.HelpFile := 'D:\Example\Application\Expo.hlp';
end;
```

Если приложение использует несколько справочных файлов, то эти файлы в зависимости от ситуации могут поочередно выбираться в качестве активного. Так, в процедурах

```
procedure TForm1.mnuHelpClick(Sender: TObject);
begin
  Application.HelpFile := 'main.hlp';
  Application.HelpCommand(HELP_FINDER, 0);
end;

procedure TForm1.mnuHelpAdditionClick(Sender: TObject);
begin
  Application.HelpFile := 'addition.hlp';
  Application.HelpCommand(HELP_FINDER, 0);
end;
```

при выборе пункта меню `mnuHelp` вызывается основной справочный файл main.hlp, а при выборе пункта меню `mnuHelpAddition` — справочный файл addition.hlp.

Функция `HelpCommand(Command: Word; Data: Longint): Boolean` обеспечивает возможность доступа к любой команде в Windows Help API. Этот метод запускает файл winhelp.exe справочной системы Windows и передает ему команду, указанную параметром `Command`, а также дополнительные данные, определяемые параметром `Data`. В результате появляется окно справочной системы Windows, с помощью которого пользователь может получить нужные сведения.

Передаваемая команда указывает тип требуемой помощи. Чаще всего используются команды:

- `HELP_FINDER` (отображается диалоговое окно **Help Topics** (Разделы помощи); параметр `Data` игнорируется);
- `HELP_KEY` (отображается тема, идентифицированная ключевым словом, адрес которого содержит параметр `Data`);
- `HELP_CONTEXT` (отображается тема, определенная значением параметра `Data`).

Функция `HelpContext(Context: THelpContext): Boolean` открывает окно справки Windows и отображает указанный параметром `Context` (целое число) экран. До вызова метода свойству `HelpFile` должно быть присвоено имя существующего справочного файла. Параметр `Context` идентифицирует справочный контекст внутри справочного файла и связан с элементами управления через одноименное свойство `HelpContext` этих элементов.

Так, в примере

```
// Для объекта-редактора следует выполнить
// инструкцию Edit1.HelpContext := 17;
// или установить это свойство через Инспектор объектов
procedure TForm1.mnuHelpEdit1Click(Sender: TObject);
begin
  Application.HelpFile := 'main.hlp';
  Application.HelpContext(17);
end;
```

при выборе пункта меню `mnuHelpEdit1` вызывается контекстная помощь для редактора `Edit1`.

Как отмечалось, любой визуальный компонент может отображать подсказку (пояснительный текст), появляющуюся при остановке на нем указателя мыши. *Текст подсказки* для каждого компонента определяется его свойством `Hint`.

По умолчанию пояснительный текст появляется через 0,5 с после остановки указателя мыши в области компонента и отображается в течение 2,5 с. Эти *временные задержки* задаются в объекте `Application` свойствами `HintPause` и `HintHidePause` типа `Integer` (в миллисекундах).

Цвет подсказки определяется свойством `HintColor` типа `TColor`. По умолчанию подсказка появляется на желтом фоне. Можно задать любой поддерживаемый системой цвет.

Замечание

Все свойства, позволяющие изменить для подсказки временные задержки и цвет фона, доступны во время выполнения приложения, и их действие распространяется на все формы приложения.

Пример установки параметров подсказки:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.HintPause := 100;
  Application.HintHidePause := 5000;
  Application.HintColor := clRed;
end;
```

Здесь значения параметров задаются при создании главной формы приложения.

Свойство `Hint` типа `String` для объекта `Application` содержит *строку подсказки* того интерфейсного элемента, на котором в данный момент времени находится указатель мыши.

Замечание

Строка подсказки для компонента содержится в свойстве `Hint` независимо от того, отображается подсказка на экране или нет.

Свойство `Icon` типа `TIcon` определяет *значок приложения*, который появляется на кнопке приложения в панели задач. Если значок не задан, по умолчанию используется зна-

чок  . Если для главной формы не установлен свой значок (через одноименное свойство `Icon`), для нее также используется значок, заданный для всего приложения. Если у главной формы есть свой значок, то в заголовке формы отображается именно он. В примере значок приложения загружается из файла printer.ico:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.Icon.LoadFromFile('printer.ico');
end;
```

Обычно для установки трех наиболее часто используемых свойств программист выбирает страницу **Application** окна параметров проекта. При задании справочного файла и значка можно воспользоваться вспомогательным окном выбора файла, нажав кнопку **Browse** (Просмотр) или **Load Icon** (Загрузить значок). Установки, сделанные в окне параметров проекта, Delphi автоматически вносит в соответствующие файлы проекта. Выбранное разработчиком изображение значка записывается в файл ресурса (res) проекта. Информация о названии приложения и файле помощи указывается в файле проекта, содержащем следующий код:

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.HelpFile := 'test.hlp';
  Application.Title := 'Проверка';
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

При выполнении длительных операций приложение не реагирует на поступающие сообщения, в том числе и на действия пользователя. Часто приложение даже не отображает в своих формах происходящие визуальные изменения, т. е. не прорисовывает формы своевременно. Такая ситуация возможна, например, в циклах обработки больших массивов данных. Исправить это положение можно путем вызова метода `ProcessMessages`, который предписывает приложению обработать поступившие и ожидающие своей очереди сообщения.

Рассмотрим в качестве примера принудительную обработку сообщений в процедуре, выполняющей отображение ряда числовых значений:

```
procedure TForm1.Button1Click(Sender: TObject);
var i: longint;
begin
  for i := 1 to 10000 do begin
    Edit1.Text := IntToStr(i);
    Application.ProcessMessages;
  end;
end;
```

После нажатия кнопки Button1 переменная i в цикле принимает ряд значений, которые отображаются в редакторе Edit1. Если из цикла убрать инструкцию Application.ProcessMessages, то после нажатия кнопки Button1 программа как бы "зависнет" на несколько секунд. При этом редактор Edit1 отобразит только последнее значение переменной i, равное 10 000. Во время выполнения цикла программа не будет реагировать на действия пользователя, связанные, например, с перемещением окна, изменением его размеров или закрытием. Отметим, что если во время выполнения цикла необходимо только своевременное отображение в поле редактора значения переменной i, то в тело цикла достаточно вставить строку, которая предписывает заново прорисовать видимую область элемента управления:

```
Edit1.Refresh;
```

Однако и в этом случае при отсутствии в теле цикла вызова метода ProcessMessages программа до окончания цикла не будет реагировать на действия пользователя.

Поэтому рекомендуется вставлять вызов метода ProcessMessages во все длительные циклы, например, возникающие при обработке большой базы данных. Если требуется досрочно завершить цикл, то в дополнение к этому методу в цикле следует разместить также операторы проверки условий прерывания цикла и выхода из цикла при их соблюдении. При этом параллельно с выполнением цикла должны производиться действия (путем обработки событий), влияющие на соблюдение условий прерывания цикла, например установка соответствующего признака при нажатии кнопки или выборе переключателя.

Рассмотрим следующий пример:

```
Var BreakFlag: boolean;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  BreakFlag := True;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  BreakFlag := False;
  for i := 1 to 10000 do begin
     Application.ProcessMessages;
     if BreakFlag then Break;
     ...
  end;
end;
```

При нажатии кнопки Button2 начинается цикл, который может быть прерван нажатием кнопки Button1, поскольку при этом переменной-признаку BreakFlag присваивается значение True. Вызов процедуры ProcessMessages в теле цикла позволяет выполнить указанное присваивание (в обработчике события нажатия кнопки Button1).

В ходе цикла приложение просто так не закрывается, даже если вызвать в теле цикла метод ProcessMessages. Одна из возможностей обеспечить *закрытие приложения* при выполнении цикла заключается в проверке свойства Terminated типа Boolean. Значение

`True` свойства означает, что приложение получило команду (сообщение) закрытия. Поэтому для закрытия приложения можно вставить в тело цикла следующую инструкцию:

```
if Application.Terminated then Break;
```

Закрыть (прекратить выполнение) приложение позволяет также метод `Terminate`, вызов которого приводит к освобождению всей занятой объектами памяти и завершению приложения. Однако обычно эта процедура явно не вызывается, а используется метод `Close` главной формы.

Объект `Application` доступен только во *время выполнения приложения*, поэтому для облегчения кодирования обработчиков событий удобно использовать компонент `ApplicationEvents`. После размещения этого компонента в форме события приложения становятся доступными также на *этапе проектирования* через Инспектор объектов, и их обработчики можно программировать так же, как обработчики других объектов, например, кнопки `Button`.

Наиболее часто используются события `OnIdle`, `OnException` и `OnHint` объекта-приложения.

Событие `OnIdle` типа `TIdleEvent` возникает при *простое* приложения. Для кодирования действий приложения, когда оно находится в режиме ожидания работы, используется обработчик события `OnIdle`. Операции, включенные в этот обработчик, выполняются каждый раз, когда приложение заканчивает текущую работу и переходит в режим простоя. Код обработчика события `OnIdle` не должен быть большим и требовать много времени на выполнение, т. к. это приведет к замедлению работы приложения в целом.

Тип `TIdleEvent` описан следующим образом:

```
type TIdleEvent = procedure(Sender: TObject; var Done: Boolean) of object;
```

Параметр `Done` указывает, как приложение реагирует на сообщения Windows, находясь в режиме ожидания. По умолчанию параметр `Done` имеет значение `True`, означающее, что обработчик события `OnIdle` не запускается до тех пор, пока не будет получено и обработано очередное сообщение. Если установить параметру `Done` значение `False`, то установленный обработчик запускается, не ожидая прихода очередного сообщения.

Процедура обработки события `OnIdle` обычно описывается и используется в модуле главной формы приложения. Например, в процедуре

```
procedure TForm1.ApplicationEvents1Idle(Sender: TObject; var Done: Boolean);
begin
  if Application.Active
    then Form1.Caption := 'Приложение активно'
    else Form1.Caption := 'Приложение неактивно';
end;
```

выполняется проверка активности приложения, что отмечается в заголовке формы `Form1`.

Если компонент `ApplicationEvents` не используется, то обработчики событий объекта-приложения можно кодировать по следующей технологии:

1. В классе какой-либо формы (обычно главной) указать заголовок процедуры-обработчика.

2. В разделе `implementation` модуля формы описать процедуру-обработчик.
3. В методе `FormCreate` создания формы назначить созданную процедуру обработчиком соответствующего события.

Указанные действия являются типовыми при написании собственных обработчиков событий, и их рассмотрение позволяет лучше понять технику обработки событий, реализованную в Delphi.

Например, при использовании события `OnIdle` для проверки активности приложения в код модуля `uAppl` формы `TForm1` нужно внести следующие изменения:

```
unit uAppl;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  // Заголовок процедуры-обработчика события OnIdle
  procedure IdleWork(Sender: TObject; var Done: Boolean);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

// Установка обработчика события OnIdle
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnIdle := IdleWork;
end;

// Обработчик события OnIdle
procedure TForm1.IdleWork(Sender: TObject; var Done: Boolean);
begin
  if Application.Active
    then Form1.Caption := 'Приложение активно'
    else Form1.Caption := 'Приложение неактивно';
end;
end.
```

Использование событий `OnException` типа `TExceptionEvent` и `OnHint` типа `TNotifyEvent` демонстрируется в *главах 7* и *8*.

При запуске приложения для него и всех его форм устанавливается *раскладка клавиатуры*, заданная в Windows по умолчанию. Пользователь выбирает нужную раскладку с помощью комбинации клавиш или значка в правом углу панели задач. Разработчик может переключать раскладку клавиатуры *программным способом*, что может быть

удобно в случае, когда, например, для приложения устанавливается русская раскладка независимо от раскладки по умолчанию Windows.

Получить список установленных для Windows раскладок клавиатуры и активизировать в приложении одну из доступных раскладок можно с помощью API-функций `GetKeyboardLayoutList` и `ActivateKeyboardLayout`.

Функция `GetKeyboardLayoutList(nBuff: integer; var List): UINT` получает список раскладок клавиатуры и заносит его в массив, указанный параметром `List`. Параметр `nBuff` задает размер (число элементов) массива, получающего список раскладок. Функция возвращает число раскладок клавиатуры, установленных на компьютере.

Функция `ActivateKeyboardLayout(hkl: HKL, Flags: UINT): HKL` устанавливает раскладку, заданную параметром `hkl`; параметр `Flags` обычно не нужен. В случае успешного выполнения функция возвращает предыдущую раскладку клавиатуры и ноль — в случае ошибки.

Рассмотрим на примере, как устанавливается раскладка клавиатуры.

```
var  rl, el: THandle;
    Layouts: array[0..7] of THandle;
          n: integer;
...
// Получение списка раскладок
procedure TForm1.Button1Click(Sender: TObject);
var i: integer;
begin
  rl := 0; el := 0;
  n := GetKeyboardLayoutList(High(Layouts) + 1, Layouts);
  for I := 0 to n - 1 do begin
    if LoWord(Layouts[i]) and $FF = Lang_Russian
       then rl := Layouts[i];
    if LoWord(Layouts[i]) and $FF = Lang_English
       then el := Layouts[i];
  end;
end;

// Установка русской раскладки
procedure TForm1.Button1Click(Sender: TObject);
begin
  if rl <> 0 then ActivateKeyboardLayout(rl, 0);
end;

// Установка английской раскладки
procedure TForm1.Button2Click(Sender: TObject);
begin
  if el <> 0 then ActivateKeyboardLayout(el, 0);
end;
```

При создании формы `Form1` (главной формы приложения) в массив `Layouts` заносится список раскладок, установленных на компьютере, а в переменных `rl` и `el` запоминаются ссылки на две раскладки. Нажатие кнопки `Button1`/`Button2` устанавливает русскую/английскую раскладку соответственно, при этом предварительно проверяется существование устанавливаемой раскладки.

Для определения раскладки, установленной по умолчанию, можно использовать свойство `DefaultKbLayout` объекта `Screen`.

Объект *Screen*

Delphi автоматически создает глобальный объект, представляющий экранную среду (объект `Screen` типа `TScreen`) для каждого приложения, поэтому обычно такие объекты программно не создаются. Использование объекта `Screen` обеспечивает возможность работы с различной экранной информацией, такой как вид указателя мыши или размеры экрана. Экранный объект, как и объект `Application`, недоступен при проектировании приложения, все его свойства и методы работают только во время выполнения программы.

Для определения *размеров экрана* можно использовать свойства `Height` и `Width` типа `Integer`, возвращающие, соответственно, высоту и ширину экранной области в пикселах. Эти свойства применяются, например, для определения установленного разрешения монитора или если необходимо масштабировать форму и ее компоненты.

Так, размещение формы в центре экрана можно реализовать с помощью следующей процедуры:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Left := (Screen.Width — Form1.Width) div 2;
  Form1.Top := (Screen.Height — Form1.Height) div 2;
end;
```

Замечание

Эту операцию можно выполнить и другим способом, а именно — установив свойство `Position` формы в значение `poScreenCenter`.

Отметим, что в Delphi у объекта экрана имеются новые свойства, позволяющие определять размеры рабочего стола (desktop): `WorkAreaRect` типа `Trect`, `WorkAreaTop`, `WorkAreaLeft`, `WorkAreaHeight` и `WorkAreaWidth` типа `Integer`.

Свойство `Cursors` типа `TCursor` определяет *вид указателя мыши* для всего приложения. Различные варианты вида указателя мыши приведены при рассмотрении свойств визуальных компонентов *(см. разд. "Свойства" главы 3)*. По умолчанию свойство `Cursors` имеет значение `crDefault`, и вид указателя мыши изменяется в зависимости от значений одноименного свойства отдельных компонентов. Если для объекта `Screen` свойство `Cursors` установить в отличное от `crDefault` значение, то указатель примет заданный вид при нахождении во всей клиентской области приложения. Так, выполнение процедуры

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.Cursor := crUpArrow;
end;
```

приведет к тому, что при нахождении указателя в клиентской области приложения он будет иметь вид вертикальной стрелки.

Замечание

Указатель выполняет свои функции независимо от его вида. С помощью щелчка кнопки мыши пользователь может, например, нажать командную кнопку, установить фокус ввода на поле редактирования или выполнить другие подобные операции. Однако задание указателю непривычного для какой-либо ситуации вида может "ввести в заблуждение".

Свойство `FormCount` типа `Integer` содержит *число форм* приложения. В это число входят все созданные формы приложения, независимо от их видимости на экране в текущий момент времени. Например, в процедуре

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(Screen.FormCount);
end;
```

при нажатии кнопки `Button1` в надписи `Label1` отображается количество созданных форм приложения.

Свойство `Forms[Index: Integer]` типа `TForm` возвращает *список форм* приложения. Это свойство представляет собой массив форм, к которым можно обращаться по индексам `Index`. При этом первая создаваемая форма доступна через `Forms[0]`, вторая — через `Forms[1]` и т. д.

Приводимая далее процедура осуществляет смену заголовка первой формы экрана:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Screen.Forms[0].Caption := 'Это первая форма приложения';
end;
```

Для определения *активной формы* в приложении можно использовать свойство `ActiveForm` типа `TForm`. Свойство доступно только для чтения и указывает форму, имеющую фокус ввода. Например:

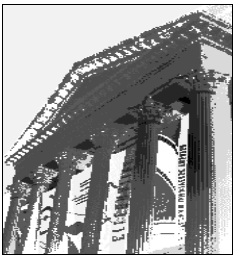
```
// Процедура является обработчиком события OnIdle приложения
procedure TForm1.IdleWork(Sender: TObject; var Done: Boolean);
begin
  Label2.Caption := Screen.ActiveForm.Name;
end;
```

Для работы со *списком шрифтов*, установленных в системе и доступных приложению, можно использовать свойство `Fonts` типа `TStrings`. Оно доступно только для чтения, с его помощью можно, например, выбрать шрифт. Так, в процедуре

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items := Screen.Fonts;
end;
```

при создании формы `Form1` приложения в список `ListBox1` загружается перечень экранных шрифтов.

Свойство `DefaultKbLayout` типа `HKL` позволяет получить раскладку клавиатуры, по умолчанию установленную в Windows. Значение этого свойства может понадобиться, например, при восстановлении раскладки по умолчанию функцией `ActivateKeyboardLayout`.



ГЛАВА 7

Обработка исключений

Исключения связаны с возникновением ошибок в процессе выполнения приложений. В этой главе рассматриваются способы обработки исключений.

Виды ошибок

В процессе разработки и выполнения программы возникают ошибки:

- синтаксические;
- логические;
- динамические.

Синтаксические ошибки вызываются нарушением синтаксиса языка, они выявляются и устраняются при компиляции программы. Их обнаруживает компилятор, выдавая сообщения и указывая в тексте программы место, где возникла ошибка. Например, в условной инструкции

```
if length(Edit1.Text) = 0 then Edit1.Text = 'NoName';
```

допущена ошибка — в записи операции присваивания отсутствует знак двоеточия (:). При ее обнаружении в ходе компиляции будет выдано соответствующее сообщение (рис. 7.1).

Логические ошибки являются следствием реализации неправильного алгоритма и проявляются при выполнении программы. Их наличие обычно не приводит к выдаче пользователю каких-либо сообщений или прекращению работы всего приложения, однако программа будет работать некорректно и выдавать неправильные результаты.

Например, рассмотрим фрагмент программы, в котором вычисляется сумма 20 элементов массива `Matrix`:

```
sum := 1;
for n := 1 to 20 do sum := sum + Matrix[n];
```

Перед началом суммирования значение суммы `sum` должно обнуляться, однако в программе допущена ошибка: вместо нуля переменной `sum` присваивается начальное зна-

чение, равное единице. Такая ошибка не приведет к прекращению выполнения программы, однако получаемый при суммировании результат будет неверен.

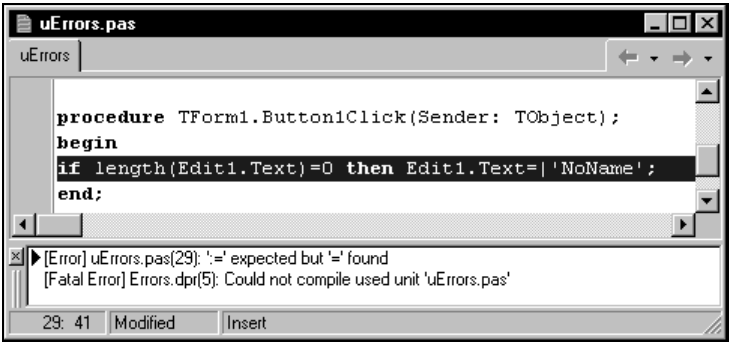


Рис. 7.1. Сообщение о синтаксической ошибке

Динамические ошибки возникают при выполнении программы и являются следствием неправильной работы инструкций, процедур, функций или методов программы, а также операционной системы. Динамические ошибки называют также *ошибками времени выполнения* (Runtime errors). Например, в инструкции присваивания

```
itog := count / number;
```

во время выполнения программы возможно появление ошибки, если переменная `number` будет иметь нулевое значение.

При отладке программ с целью выявления динамических ошибок удобно задавать отладочное (пошаговое или трассировочное) выполнение программы с использованием окон просмотра (**Watch List**). В окне просмотра можно указать выражения, имена переменных или свойств объекта, изменение значений которых требуется проконтролировать.

Для отладочного выполнения программы с помощью меню **Run** (Выполнение) можно использовать следующие варианты:

- команда **Step Over** (Шаг с обходом) предписывает выполнение одной строки кода программы с обходом процедур (процедура выполняется как единый модуль);
- команда **Trace Into** (Трассирование до) предписывает выполнение одной строки кода программы с заходом в процедуры и их последующим построчным выполнением;
- команда **Trace To Next Source Line** (Трассирование до следующей строки кода) предписывает выполнение программы с остановкой на следующей выполнимой строке кода программы;
- команда **Run To Cursor** (Выполнение до курсора) задает выполнение загруженной программы до места размещения курсора в Редакторе кода;
- команда **Run Until Return** (Выполнение до возврата) задает выполнение программы до момента возврата из текущей процедуры.

Для добавления очередного контролируемого выражения в окно просмотра служит команда **Run | Add Watch**. После ее выполнения открывается диалоговое окно **Watch Properties** (Свойства просмотра) для задания свойств контролируемого выражения (рис. 7.2). В поле **Expression** (Выражение) задается выражение для контроля его значения, в поле **Group name** (Имя группы) можно задать имя группы, в которую будет помещено контролируемое выражение. С помощью группы переключателей в нижней части диалогового окна выбирается формат отображения значения контролируемого выражения.

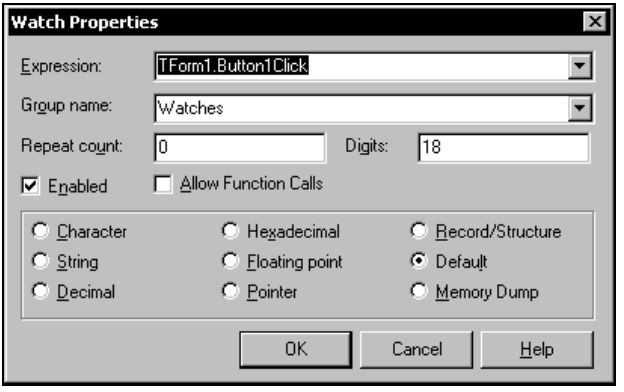


Рис. 7.2. Диалоговое окно свойств контролируемого выражения

После нажатия кнопки **OK** открывается окно **Watch List** (Список просмотра) (рис. 7.3) с добавленным к вновь созданному или существующему списку новым контролируемым выражением. Имена вкладок в окне **Watch List** соответствуют именам созданных групп просмотра. Для задания отображения окна **Watch List** (без добавления нового контролируемого выражения) служит команда **View | Debug Windows | Watches** (Вид | Окна отладки | Просмотры).

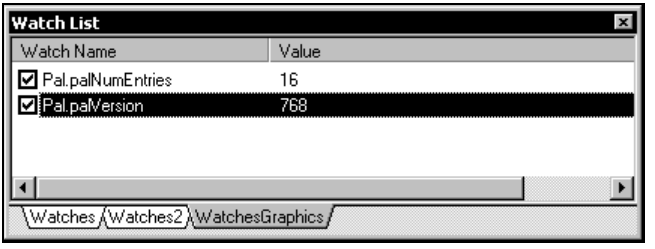


Рис. 7.3. Список контролируемых выражений в окне **Watch List**

Замечание

Возможность задания групп для помещения в них списков контролируемых выражений, заметно повышающая удобство работы с их элементами, впервые появилась в Delphi 7.

Программист должен предвидеть возможность возникновения динамических ошибок и предусматривать их обработку. Для обработки динамических ошибок введено понятие *исключения*, которое представляет собой нарушение условий выполнения программы,

вызывающее прерывание или полное прекращение ее работы. Обработка исключения состоит в нейтрализации вызвавшей его динамической ошибки.

Исключения могут возникать по различным причинам, например, в случае нехватки памяти, из-за ошибки преобразования, в результате выполнения вычислений и т. д. В любом случае независимо от источника ошибки приложение информируется (получает сообщение) о его возникновении. Исключение остается актуальным до тех пор, пока не будет обработано глобальным обработчиком или локальными процедурами.

В Delphi для обработки динамических ошибок в выполняемый файл приложения встраиваются специальные фрагменты кода, предназначенные для реагирования на исключения. Механизмы обработки ошибок в Delphi инкапсулированы в класс `Exception`. Разнообразные потомки этого класса охватывают значительное число ошибок, которые могут возникнуть в среде Windows.

Возникающие при выполнении программы динамические ошибки автоматически преобразуются средствами Delphi в соответствующие объекты-исключения. Объект-исключение содержит информацию о типе ошибки и при возникновении исключения заставляет программу или ее поток (составляющую процесса) временно приостановиться. После обработки исключения объекты-исключения автоматически удаляются.

Классы исключений

Как уже сказано, исключения в Delphi являются объектами. Базовым классом для всех исключений служит класс `Exception`, описываемый в модуле `SysUtils`. Потомки этого класса содержат большое количество исключений, которые могут возникнуть в процессе выполнения приложения. Любые новые классы исключений должны создаваться как потомки класса `Exception`, что обеспечивает возможность их распознавания и обработки как исключений.

В отличие от других классов, имена базового класса `Exception` и его потомков, связанных с исключениями, начинаются не с буквы `T`, а с буквы `E`. Выполнять это соглашение не обязательно, но следовать ему — правило хорошего стиля.

Класс `Exception` происходит непосредственно от класса `Object` и имеет достаточно простое описание, содержащее 2 свойства и 8 методов. Так как объекты-исключения создаются при выполнении приложения, их свойства также доступны только в это время.

Свойство `Message` типа `String` содержит описание исключения. Часто при возникновении исключения этот текст появляется в диалоговом окне глобального обработчика исключений.

Свойство `HelpContext` типа `THelpContext` хранит уникальный номер (идентификатор контекста), указывающий на раздел контекстной помощи для объекта-исключения. Если этот номер отличается от нуля, то при вызове контекстной помощи отображается соответствующий раздел справки. Если же он равен нулю (по умолчанию), то отображается раздел справки родителя этого объекта.

Наиболее часто используемым методом объекта-исключения является конструктор `Create(const Msg: String)`, который создает объект-исключение. Строка `Msg` указывает текст для свойства `Message` создаваемого объекта.

Аналогичное назначение имеет конструктор `CreateFmt(const Msg: String; const Args: array of const)`, дополнительно позволяющий задать текст сообщения в форматированном виде.

Методы `CreateHelp(const Msg: String; AHelpContext: Integer)` и `CreateFmtHelp(const Msg: String; const Args: array of const; AHelpContext: Integer)` помимо создания объекта-исключения обеспечивают задание для него номера, указывающего раздел контекстной помощи (свойство `HelpContext`).

Как отмечено ранее, у класса `Exception` много потомков, каждый из которых предназначен для обработки конкретной ошибки. Некоторые классы производятся непосредственно от класса `Exception`, например класс `EAbort`:

```
type EAbort = class(Exception);
```

Ряд классов происходит от потомков класса `Exception`. Например, класс `EMathError`, являющийся базовым классом для исключений при операциях с плавающей точкой, производится от класса `EExternal` — непосредственного потомка класса `Exception`. В свою очередь, от класса `EMathError` происходят несколько классов, обеспечивающих обработку конкретных ошибок, возникающих при выполнении операций с плавающей точкой:

```
type
EExternal = class(Exception)
  public
    ExceptionRecord: PExceptionRecord;
  end;
EMathError  = class(EExternal);
EInvalidOp  = class(EMathError);
EZeroDivide = class(EMathError);
EOverflow   = class(EMathError);
EUnderflow  = class(EMathError);
```

Наиболее часто используются следующие классы исключений:

- `EAbort` — "тихое" исключение, используемое для прерывания текущего блока кода без вызова глобального обработчика;
- `EOutOfMemory` — нехватка оперативной памяти для выполнения операции;
- `EInOutError` — ошибка ввода/вывода файла любого типа;
- `EIntError` — базовый класс для ошибок, связанных с целочисленными вычислениями; специализированные исключения обрабатываются потомками этого класса:
  - `EIntOverflow` — переполнение в операции с целочисленными переменными;
  - `EDivByZero` — деление целого числа на ноль;
  - `ERangeError` — присвоение целочисленной переменной значения, выходящего за пределы допустимого диапазона; может возникать при попытке обращения к элементам массива по индексу, выходящему за границы;
- `EMathError` — базовый класс для ошибок в операциях с плавающей точкой; специализированные исключения обрабатывают потомки этого класса:

- `EInvalidOp` — ошибка в операции над числом с плавающей точкой; исключение может возникать по различным причинам, например, процессор пытается выполнить недействительную операцию или произошло переполнение стека;
- `EZeroDivide` — деление на ноль числа с плавающей точкой;
- `EOverFlow` — присвоение вещественной переменной значения, которое не помещается в отведенной области памяти;
- `EUnderflow` — потеря значимости при операциях над числами с плавающей точкой, результат получает нулевое значение;

- `EInvalidPointer` — некорректная операция с указателем;
- `EInvalidCast` — неверное приведение типов с помощью инструкции `as`;
- `EConvertError` — ошибка преобразования типа, возникающая, например, при преобразовании строковых данных в числовые с помощью функции `StrToInt` или `StrToFloat`;
- `EFCreateError` — ошибка создания файла;
- `EFOpenError` — ошибка открытия файла;
- `EResNotFound` — в указанном файле отсутствует ресурс;
- `EListError`, `EStringListError` — ошибки в списках;
- `EPrinter` — ошибка печати;
- `EMenuError` — ошибка в меню приложения (часто возникает при динамической настройке меню, например, при создании нового пункта, имя которого уже существует);
- `EInvalidGraphicOperation` — неправильная операция с графическим объектом.

Замечание

В Delphi 7 добавлен класс исключений `EFileStreamError`, предназначенный для обработки общих ошибок потока. Исключение имеет место в случае возникновения ошибок при передаче потоков данных. От этого класса происходят классы `EFileStreamError` и `EFOpenError`. Новый класс `EFileStreamError` может принимать параметр `FileName`, в результате текст сообщения исключения теперь содержит имя файла, в котором произошла ошибка.

Потомки класса `Exception` могут добавлять к нему дополнительные свойства, конкретизирующие связанные с этими классами исключения.

Обработка исключений

Для обработки исключений в приложении есть один *глобальный* обработчик и несколько специализированных процедур-обработчиков, реагирующих на соответствующие исключения. Каждое исключение обрабатывает свой специализированный *локальный* обработчик. Исключение, не имеющее своего локального обработчика, обрабатывается глобальным обработчиком приложения.

Далее подробно рассматривается обработка различных исключений и приводятся соответствующие примеры. В примерах других глав обработка исключений, как правило, не предусматривается с целью сокращения текста листингов.

Глобальная обработка

Механизм глобальной обработки исключений реализуется через объект `Application`, который есть в любом приложении. Создание и запуск объекта `Application` можно увидеть в файле проекта (dpr). Одна из функций объекта `Application` — обеспечение приложения глобальным обработчиком исключений.

При получении от операционной системы сообщения об исключении объект `Application` генерирует событие `OnException` типа `TexceptionEvent`, обработчик которого и является глобальным обработчиком исключений. По умолчанию на это событие для всех видов динамических ошибок, не имеющих своего обработчика, реагирует метод `HandleException` приложения. В теле процедуры `HandleException(Sender: TObject)` вызывается метод `ShowException` приложения, выводящий на экран диалоговое окно с описанием возникшего исключения (рис. 7.4). Такая обработка не устраняет причину исключения, но обеспечивает пользователя информацией об ошибке и облегчает ее поиск и устранение.

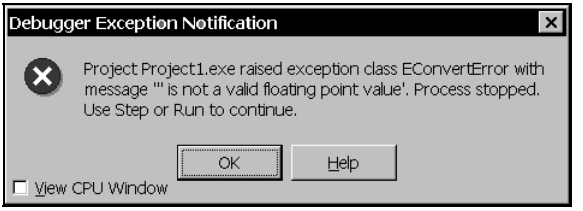


Рис. 7.4. Окно с сообщением об ошибке

Программист может выполнить более полную обработку исключений, создав собственный глобальный обработчик события `OnException`. Для этого удобно использовать компонент `ApplicationEvents`.

Событие `OnException` имеет тип `TExceptionEvent`, который описан следующим образом:

```
type TExceptionEvent = procedure(Sender: TObject; E: Exception) of object;
```

Параметр `E` представляет собой объект-исключение. С его помощью можно получить доступ к свойствам объекта-исключения, например, к свойству `Message`, содержащему описание возникшего исключения.

Рассмотрим в качестве примера процедуру — глобальный обработчик, реализующую собственную обработку исключения:

```
procedure TForm1.ApplicationEvents1Exception(Sender: TObject; E: Exception);
begin
  MessageDlg(E.Message, mtError, [mbOK], 0);
  // Здесь размещаются инструкции, выполняющие
  // развитую обработку исключения
end;
```

При возникновении исключения для пользователя выводится диалоговое окно с сообщением об ошибке. Реакция на ошибку со стороны приведенной процедуры в данном случае не отличается от реакции устанавливаемого по умолчанию глобального обработчика и заключается просто в кратком информировании пользователя об ошибке. В более сложном случае глобальный обработчик может содержать код, зависящий от особенностей конкретной программы, например, освобождающий память или закрывающий рабочие файлы.

Локальная обработка

Чтобы сделать возможным использование локальных (специализированных) обработчиков исключений, в состав языка введены две конструкции: `try..finally` и `try..except`. Обе конструкции имеют похожий синтаксис, но разное назначение. Блоки `try` включают в себя инструкции программы (например, запись на диск или преобразование данных), при выполнении которых может возникнуть исключение.

Выбор конструкции зависит от применяемых инструкций программы и действий, выполняемых при возникновении ошибки. Конструкции `try` могут содержать одну или более инструкций, а также быть вложенными друг в друга.

Конструкция `try..finally` состоит из двух блоков (`try` и `finally`) и имеет следующую форму:

```
try
  // Инструкции, выполнение которых может вызвать ошибку
finally
  // Инструкции, которые должны быть выполнены даже в случае ошибки
end;
```

Она применяется для выполнения всех необходимых действий перед передачей управления на следующий уровень обработки ошибки или глобальному обработчику. Такими действиями могут являться, к примеру, освобождение оперативной памяти или закрытие файла. Эта конструкция не обрабатывает объект-исключение и не удаляет его, а выполняет действия, которые должны быть произведены даже в случае возникновения ошибки.

Конструкция `try..finally` работает так: если в любой из инструкций блока `try` возникает исключение, то управление передается первой инструкции блока `finally`. Если же исключение не возникло, то последовательно выполняются все инструкции обоих блоков.

Рассмотрим следующий пример:

```
procedure TForm1.btnShowClick(Sender: TObject);
var MyPicture: TBitmap;
begin
  try
    // Создание объекта Ris типа "растровое изображение"
    MyPicture := TBitmap.Create;
    // Загрузка изображения из указанного графического файла
    // в объект MyPicture
    MyPicture.LoadFromFile('photo.bmp');
```

```
    // Вывод изображения из объекта MyPicture на поверхность формы
    Form1.Canvas.Draw(10, 20, MyPicture);
  finally
    // Даже в случае ошибки при выполнении двух предыдущих операций
    // уничтожить объект MyPicture
    MyPicture.Free;
  end;
end;
```

При нажатии кнопки `btnShow` в форме отображается изображение из файла photo.bmp. Предварительно изображение загружается из файла в промежуточный графический объект `MyPicture`. После вывода изображения на поверхность формы объект `MyPicture` уничтожается. Операция уничтожения объекта `MyPicture` и освобождения занимаемой им памяти должна быть выполнена в любом случае, поэтому инструкция `MyPicture.Free` расположена в блоке `finally`. Инструкции же, выполнение которых чревато исключением, расположены в блоке `try`. Исключение может произойти, например, при создании объекта `Ris` или при загрузке изображения из файла.

Так как конструкция `try..finally` не ликвидирует исключительную ситуацию, в приведенной процедуре при возникновении исключения глобальный обработчик выдаст сообщение о характере ошибки.

Конструкция `try..except` также состоит из двух блоков (`try` и `except`) и имеет следующую форму:

```
try
   {Инструкции, выполнение которых может вызвать ошибку}
except
   {Инструкции, которые должны быть выполнены в случае ошибки}
end;
```

В отличие от предыдущей, данная конструкция применяется для перехвата исключения и предоставляет возможность его обработки.

Конструкция `try..except` работает так: если в инструкциях блока `try` возникает исключение, то управление передается первой инструкции блока `except`. Если же исключение не возникло, то инструкции блока `except` не выполняются. При появлении исключения инструкции блока `except` могут ликвидировать исключительную ситуацию и восстановить работоспособность программы. Для исключений, обрабатываемых в конструкции `try..except`, глобальный обработчик не вызывается, а обработку ошибок должен обеспечить программист.

Рассмотрим пример локальной обработки исключения с помощью конструкции `try..except`:

```
procedure TForm1.btnOpenClick(Sender: TObject);
begin
  try
    if OpenDialog1.Execute then begin
      Table1.Active := False;
      Table1.TableName := OpenDialog1.FileName;
      Table1.Active := True;
    end;
```

```
  except
    MessageDlg('Ошибка открытия таблицы ', OpenDialog1.FileName, '!',
               mtError, [mbOK], 0);
  end;
end;
```

При нажатии кнопки `btnOpen` появляется окно выбора файла таблицы базы данных для открытия. После выбора главного файла таблицы набор данных `Table1` связывается с этой таблицей, и выполняется открытие набора данных. При какой-либо ошибке, например, в случае выбора файла, не являющегося главным файлом таблицы, может возникнуть исключение. Поэтому инструкции, управляющие выбором файла и открытием набора данных, включены в блок `try`. При возникновении исключения его обрабатывают инструкции блока `except`. В данном примере обработка заключается просто в выдаче предупреждающего сообщения. Если выбор файла и открытие набора данных выполнены корректно, то инструкции блока `except` не выполняются.

Блок `except` можно разбить на несколько частей с помощью конструкций `on..do`, позволяющих анализировать класс исключения для его более удобной и полной обработки. Конструкция `on..do` применяется в случаях, когда действия по обработке исключения зависят от класса исключения, и имеет следующую форму:

```
on {Идентификатор: класс исключения} do
   {Инструкции обработки исключения этого класса};
else {Инструкции};
```

В инструкции `on` класс возникшего исключения сравнивается с указанным классом исключения. В случае совпадения классов выполняются инструкции после слова `do`, реализующие обработку этого исключения.

Идентификатор (произвольное имя, заданное программистом) является необязательным элементом и может отсутствовать, при этом не ставится и разделительный знак двоеточия (`:`). Идентификатор — это локальная переменная, представляющая собой экземпляр класса исключения, который можно использовать для доступа к объекту возникшего исключения. Эта переменная доступна только внутри "своей" конструкции `on..do`.

Если класс возникшего исключения не совпадает с проверяемым классом, то выполняются инструкции после слова `else`. Блок `else` является необязательным и может отсутствовать.

Если в блоке `except` расположено несколько конструкций `on..do`, то `else` располагается в конце блока и относится ко всей совокупности конструкций `on..do`. Следующая после слова `else` инструкция выполняется в том случае, если обработка исключения не была осуществлена ни одной из инструкций, расположенных в любой из конструкций `do` блока. Инструкции, следующие после слов `do` и `else`, могут быть составными.

Несколько последовательных конструкций `on..do` по структуре и логике обработки напоминают инструкцию выбора `case..of`. Обычно в операторные скобки `on..do` берутся исключения, появление которых наиболее вероятно.

Рассмотрим следующий пример:

```
procedure TForm1.btnAddClick(Sender: TObject);
var x, y, res :real;
```

```
begin
  try
    x := StrToInt(Edit1.Text);
    y := StrToInt(Edit2.Text);
    res := x / y;
    Edit3.Text := FloatToStr(res);
  except
    on EZeroDivide do begin
       MessageDlg('Попытка деления на ноль!', mtError, [mbOK], 0);
       Edit2.SetFocus;
       Edit3.Text := 'Ошибка!';
    end;
    on EO: EConvertError do begin
       MessageDlg('Ошибка преобразования!' +
                  #10#13 + EO.Message, mtError, [mbOK], 0);
       Edit1.SetFocus;
       Edit3.Text := 'Ошибка!';
    end;
  else begin
    MessageDlg('Ошибка не идентифицирована!', mtWarning, [mbOK], 0);
    Edit1.SetFocus;
    Edit3.Text := 'Ошибка!';
  end;
end;
```

В поля `Edit1` и `Edit2` вводятся два целых числа. При нажатии кнопки `btnAdd` выполняется преобразование этих чисел из текстового формата в числовой, после чего первое число делится на второе, а результат помещается в поле `Edit3`. Выполняющие эти действия инструкции могут вызвать ошибку, поэтому они помещены в блок `try`. Если возникает исключение, оно анализируется в блоке `except`. Проверяются следующие варианты: `EConvertError` (ошибка преобразования строкового типа в целочисленный) и `EZeroDivide` (деление на ноль). При этом используются две конструкции `on..do`.

При возникновении одного из прогнозируемых исключений выдается сообщение об исключении. Любое другое исключение не будет опознано. В этом случае выполняются инструкции блока `else`. В результате выдается предупреждение о неопознанной ошибке.

В конструкции `on..do`, применяемой для анализа исключения на ошибку преобразования, использован идентификатор `EO` (ссылка на объект-исключение), позволяющий обращаться к объекту по имени. Например, для доступа к сообщению о характере ошибки необходимо указать `EO.Message`.

Кроме выдачи сообщений, в блоке `except` выполняются такие действия, как установка фокуса ввода на определенное поле редактирования и помещение в поле результата строки `Ошибка!`.

Конструкции `try` могут быть *вложенными* и размещаться одна в другой. При этом внешняя и внутренняя конструкции могут иметь любой из двух рассмотренных видов. Обязательным условием является то, что внутренний блок должен полностью размещаться во внешнем блоке.

Например:

```
try
   {Инструкции}
   try
      {Инструкции}
   finally
      {Инструкции}
   end;
except
   {Инструкции}
end;
```

или

```
try
   {Инструкции}
   try
      {Инструкции}
   except
      {Инструкции}
   end;
finally
   {Инструкции}
end;
```

Если какие-либо действия должны быть выполнены независимо от того, произошла ошибка или нет, то удобно использовать конструкцию `try..finally`. Однако, как отмечалось, эта конструкция не обрабатывает исключение, а лишь в некоторой степени смягчает его последствия. Если же требуется произвести и локальную обработку исключения, то можно включить конструкцию `try..finally` в конструкцию `try..except`. При возникновении исключения это позволяет выполнить обязательные инструкции блока `finally` и обработать исключение инструкциями блока `except`. Например:

```
procedure TForm1.btnShowClick(Sender: TObject);
var MyPicture: TBitmap;
begin
  try
    // Инструкции, которые могут вызвать ошибку
    try
      MyPicture := TBitmap.Create;
      MyPicture.LoadFromFile('photo.bmp');
      Form1.Canvas.Draw(10, 20, MyPicture);
    // Обязательное освобождение памяти
    finally
       MyPicture.Free;
    end;   {try .. finally}
  // Анализ и обработка ошибки
  except
    on EFOpenError do
       MessageDlg('Ошибка открытия файла photo.bmp!',
                  mtError, [mbOK], 0);
```

```
    on EInOutError do
       MessageDlg('Ошибка чтения файла photo.bmp!', mtError, [mbOK], 0);
    on EInvalidGraphicOperation do
       MessageDlg('Ошибка графики!', mtError, [mbOK], 0);
    on EInvalidGraphic do
       MessageDlg('Неправильный графический формат файла!',
                  mtError, [mbOK], 0);
    else Application.HandleException(Sender);
  end;   {try .. except}
end;
```

При нажатии кнопки `btnShow` в форме отображается изображение из файла photo.bmp. В отличие от ранее рассмотренного примера, конструкция `try..finally` вложена в конструкцию `try..except`, в которой выполняются анализ и обработка возможного исключения. В блоке `except` проверяется, к какому классу относится возникшее исключение: ошибка открытия файла, ошибка чтения файла, ошибка выполнения графической операции или неправильный формат графического файла. Обработка возникающих исключений заключается в выдаче пользователю сообщения о характере ошибки. Если исключение не соответствует ни одному из проверяемых классов, то вызывается глобальный обработчик исключений с помощью инструкции `Application.HandleException(Sender)`, расположенной после `else`.

Аналогичным образом программируется вложение конструкции `try..except` в конструкцию `try..finally`.

Такие компоненты, как наборы данных `Table` и `Query`, используемые при работе с базами данных, имеют события, генерируемые при возникновении ошибок модификации данных:

- `OnEditError` (ошибка редактирования записи);
- `OnUpdateError` (ошибка обновления записи);
- `OnDeleteError` (ошибка удаления записи).

Эти события также можно использовать для локальной обработки исключений.

Вызов исключений

При необходимости исключение можно сгенерировать программно. Для этого используется инструкция `raise`, которая создает объект-исключение — экземпляр класса `Exception` или его потомок. Инструкция `raise` имеет следующий синтаксис:

```
Raise  ClassException.Method;
```

Здесь `ClassException` является классом исключения, на основе которого создается объект-исключение, а конструктор `Method` выполняет создание объекта-исключения. Через этот метод объекту-исключению передаются все данные, несущие информацию о виде исключения. Для создания объектов-исключений чаще всего используются методы `Create` и `CreateFmt` классов исключений.

Код, следующий после слова `raise`, аналогичен коду, используемому при создании нового экземпляра класса, однако особенностью синтаксиса инструкции `raise` является отсутствие оператора присваивания.

Пример генерации исключения класса Exception:

```
Raise Exception.Create('Новая ошибка!);
```

В качестве текста, поясняющего возникшее исключение, просто передается строка Новая ошибка!.

После вызова исключения автоматически создается экземпляр указанного класса ошибки, который существует до момента окончания обработки исключения, после чего автоматически уничтожается. То есть исключение, инициированное программным способом, обрабатывается так же, как другие исключения.

Рассмотрим на примере генерацию и обработку исключения:

```
function TForm1.TestEdit(EditStr: string; Pr: boolean): string;
begin
  if length(EditStr) > 5 then
    Raise Exception.Create(EditStr + ' — слишком длинная строка!');
  if  Pr then Result := EditStr + '-' else Resultc := EditStr;
end;

procedure TForm1.btnAddClick(Sender: TObject);
begin
  try
    Edit1.Text := TestEdit(Edit1.Text, True);
    Edit2.Text := TestEdit(Edit2.Text, False);
    Edit3.Text := Edit1.Text + Edit2.Text;
  except
    on EO: Exception do begin
      MessageDlg(EO.Message, mtError, [mbOK], 0);
      Edit3.Text := 'Ошибка!';
    end;
  else begin
        MessageDlg('Ошибка не идентифицирована.',
                   mtWarning, [mbOK], 0);
        Edit3.Text := 'Ошибка!';
      end;
  end;
end;
```

При нажатии кнопки btnAdd в поле редактирования Edit3 выводится строка, являющаяся результатом соединения двух строк, взятых из полей редактирования Edit1 и Edit2. К тексту поля Edit1 автоматически добавляется разделитель (-). Обязательным является требование, чтобы длина каждой из исходных строк не превышала пяти символов. Для проверки длины текста использована функция TestEdit, вызываемая дважды из обработчика события OnClick кнопки btnAdd. При выполнении функции TestEdit может возникать исключение, поэтому ее вызовы помещены внутрь конструкции try..except.

В качестве входного параметра функция TestEdit получает строку EditStr. Если ее длина превышает пять символов, то программно генерируется исключение Exception, и из функции TestEdit управление передается блоку except вызывающей процедуры btnAddClick. При генерации исключения расположенная после инструкции raise

условная инструкция не выполняется. Таким образом, инструкция `raise` действует как процедура раннего выхода `exit`, но, в отличие от нее, конструкция `raise` позволяет удобно передавать в вызывающую процедуру информацию о характере ошибки.

Обработка исключения заключается в выдаче соответствующего сообщения о характере ошибки.

Инструкцию `raise`, кроме генерации исключения, можно использовать и для более удобной обработки исключений, вызванных *ранее*. В этом случае ее синтаксис совсем прост:

```
Raise;
```

Если инструкция `raise` стоит в блоке `except` конструкции `try..except`, то она предписывает не обрабатывать возникшее исключение, а передать его на более высокий уровень обработки: внешней конструкции `try..except` или глобальному обработчику ошибок.

Рассмотрим в качестве примера обработку ошибок с использованием конструкции `try..except`, вложенной в конструкцию `try..finally`.

```
procedure TForm1.btnDivClick(Sender: TObject);
var x, y, res : integer;
            f : system.text;
begin
  try
    System.Assign(f, 'DivResult.txt'); Rewrite(f);
    x := StrToInt(Edit1.Text);
    y := StrToInt(Edit2.Text);
    try
      res := x div y;
      Edit3.Text := IntToStr(res);
      Writeln(f, 'Результат = ', res);
    except
      on EDivByZero do
         MessageDlg('Деление на ноль', mtError, [mbOK], 0);
      else Raise;
    end;
  finally
    System.Close(f);
  end;
end;
```

При нажатии кнопки `btnDiv` выполняется целочисленное деление чисел, помещенных в поля редактирования `Edit1` и `Edit2`. Результат деления помещается в поле `Edit3` и записывается в файл DivResult.txt.

В блоке `try` внешней конструкции открывается файл для записи результата и содержимое полей редактирования преобразуется в целые числа. Если при этом возникает исключение, то выполняется процедура закрытия файла, расположенная в блоке `finally`.

При отсутствии ошибок на предыдущем этапе выполняются инструкции блока `try` внутренней конструкции `try..except`. Если в ходе выполнения одной из этих инструкций возникает исключение, то управление передается блоку `except`. При обработке ис-

ключения проверяется его класс, и если он совпадает с `EDivByZero` (наиболее вероятное исключение для этого примера), то локальный обработчик выдает предупреждение о попытке деления на ноль.

При возникновении исключения другого класса инструкция `raise` передает его для обработки во внешнюю конструкцию `try`. Так как блок `finally` фактически не обрабатывает ошибку, а смягчает и компенсирует ее последствия, то в итоге исключение обрабатывается глобальной процедурой приложения.

Отметим, что процедура закрытия файла выполняется независимо от вида и места возникновения ошибок.

Наряду с рассмотренными вариантами использования инструкция `raise` имеет еще одну особенность, связанную с ее совместным применением с исключением класса `EAbort`.

Класс `EAbort` и его потомки представляют собой так называемые *"тихие исключения"*. Эти классы применяются в случаях, когда при выполнении какого-либо фрагмента кода, например тела цикла, возникает ошибка, и дальнейшее выполнение блока, в котором расположены инструкции, следует *прекратить*. То есть требуется выполнить ранний возврат, скажем, из процедуры или функции, чтобы сократить процесс выполнения программы и упростить обработку данных.

Ранний возврат можно выполнить также путем генерирования исключения, класс которого отличается от класса исключения `EAbort` (такой пример рассмотрен ранее). Однако именно "тихие исключения" при их возникновении не проходят дальше первого уровня обработки. В случаях, когда нежелательно, чтобы ошибка, вызвавшая прерывание выполнения инструкций, обрабатывалась локальным или глобальным обработчиками, можно создать исключение класса `EAbort`, например, так:

```
Raise EAbort.Create('Описание возникшей ситуации (ошибки)');
```

Функциональным эквивалентом приведенной инструкции `raise` является использование процедуры `Abort`, входящей в модуль `SysUtils`.

Замечание

Эти способы создания "тихого исключения" отличаются тем, что в инструкции `raise` можно задать текст, поясняющий характер ошибки. Однако надо отметить, что при генерации "тихого исключения" пользователю этот текст не выдается.

Пример прерывания выполнения цикла:

```
var pBreak: boolean;
procedure TForm1.Button1Click(Sender: TObject);
var i: integer;
begin
  pBreak := False;
  Table1.First;
  for i := 1 to Table1.RecordCount do begin
    Edit1.Text := IntToStr(i);
    Edit1.Refresh;
    // Здесь располагаются инструкции, выполняющие обработку i-й записи
    Application.ProcessMessages;
```

```
    if pBreak then Raise EAbort.Create('Обработка прервана.');
    Table1.Next;
  end;
end;
```

В приведенной процедуре тело цикла выполняется до тех пор, пока логическая переменная `pBreak` имеет значение `False`. В цикле осуществляются перебор и обработка всех записей набора данных `Table1`, начиная с первой. В поле редактирования `Edit1` отображается номер обрабатываемой записи. Если изменить значение переменной на `True`, то в условной инструкции будет сгенерировано "тихое исключение", и цикл, а вместе с ним и процедура `Button1Click`, прервутся. При этом пользователю не выдается никаких сообщений. Если для генерации "тихого исключения" использовать процедуру `Abort`, то инструкция прерывания цикла имеет следующий вид:

```
if pBreak then Abort;
```

Для изменения значения переменной `pBreak` можно, например, использовать кнопку `Button2`, обработчик события нажатия которой имеет вид:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  pBreak := True;
end;
```

Для того чтобы при выполнении цикла по перебору записей набора данных `Table1` могли обрабатываться происходящие в это время события, в цикл вставлен вызов процедуры `ProcessMessages` объекта `Application`. Вызов этой процедуры предписывает обработать находящиеся в очереди события, не дожидаясь завершения обработки текущего события, в данном случае нажатия кнопки `Button1`.

Вызов "тихого исключения" можно использовать также для блокирования перевода набора данных в режим редактирования. С этой целью в обработчик события `BeforeEdit`, возникающего непосредственно перед переводом набора данных в режим редактирования, вставляется условная инструкция. В этой инструкции можно задать проверку условий, при выполнении (или невыполнении) которых генерируется "тихое исключение", возвращающее набор данных в прежний режим, чаще всего в режим просмотра.

Например, в процедуре

```
procedure TForm1.Table1BeforeEdit(DataSet: TDataSet);
begin
  if CheckBox3.Checked then
     Raise EAbort.Create('Редактирование данных запрещено!');
end;
```

для анализа возможности редактирования набора данных используется состояние флажка `CheckBox3`. Установленный флажок соответствует запрету изменения данных. Флажок `CheckBox3` может иметь заголовок **Редактирование запрещено** или **Только просмотр**.

Как отмечалось, при генерации "тихого исключения" процесс прерывается, но пользователь не получает никаких сообщений. Если нужно проинформировать пользователя о возникшей ситуации, то вместо "тихого исключения" можно генерировать обычное исключение, например, так:

```
procedure TForm1.Table1BeforeEdit(DataSet: TDataSet);
begin
  if CheckBox3.Checked then
     Raise Exception.Create('Редактирование данных запрещено!');
end;
```

При работе с базами данных программно сгенерировать исключение можно описанными выше способами. Кроме того, есть специальные методы. В частности, процедура `DatabaseError(const Message: string; Component: TComponent = nil)` генерирует исключение `EDatabaseError`. Параметр `Message` содержит текст, описывающий характер ошибки. Например:

```
DatabaseError('Ошибка EDatabaseError ');
```

Обработку возбуждаемого исключения `EDatabaseError` можно в дальнейшем выполнить обычным способом.

Процедура `DBIError(ErrorCode: DBIResult)` генерирует исключение `EDBEngineError`. Параметр `ErrorCode` содержит код ошибки. Пример вызова исключения `EDBEngineError` программным способом:

```
DBIError($21);
```

Здесь генерируется исключение `EDBEngineError` с шестнадцатеричным кодом 21. Последующая обработка исключения производится стандартными методами.

Создание классов исключений

Новое исключение можно задать программно, для чего требуется описать соответствующий класс, порождаемый непосредственно от базового класса `Exception` или любого его потомка. Тогда вводимое исключение будет наследовать свойства и методы, требуемые для поддержки рассмотренного ранее механизма обработки исключений. Обычно новый класс исключения предоставляет более подробную информацию об ошибке.

Чтобы создать класс исключения, в разделе `interface` модуля нужно описать новый класс, введя в него новые или переопределив старые поля, свойства и методы. Затем эти новые или измененные элементы класса определяются требуемым образом в разделе `implementation`.

После создания класса исключения соответствующее исключение может быть вызвано инструкцией `raise` и обработано с помощью локальных и глобального обработчиков, как и любое другое исключение.

В листинге 7.1 в качестве примера приводятся фрагменты модуля с описанием формы `Form1`, связанные с новым классом исключения `EExceptionCode`.

Листинг 7.1. Создание нового класса исключения `EExceptionCode`

```
interface
// Новый класс исключения (EExceptionCode)
type EExceptionCode = class(Exception)
     private
        FErrCode: integer;
     public
        property ErrCode: integer read FErrCode write FErrCode;
        constructor Create(const Msg: string; ErrorCode: integer);
     end;
// Новый класс для глобальной обработки исключений
type
  TGlobalHandler = class
  public
    procedure ProcExcept(Sender: TObject; EInstance: Exception);
  end;
var GlobalHandler: TGlobalHandler;
...
implementation

// Переопределение конструктора для нового класса EExceptionCode
constructor EExceptionCode.Create(const Msg: string; ErrCode: integer);
begin
  Inherited Create (Msg);
  FErrCode := ErrCode;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  // Создание нового класса для глобальной обработки исключений.
  // В файле проекта должна быть ссылка на процедуру в составе этого класса
  GlobalHandler := TGlobalHandler.Create;
end;

// Описание процедуры — глобального обработчика исключений
procedure TGlobalHandler.ProcExcept(Sender: TObject; EInstance: Exception);
begin
  if EInstance is EExceptionCode then
    case (EInstance as EExceptionCode).ErrCode of
      19: MessageDlg('В поле Edit1 или Edit2 не число!', mtError, [mbOK], 0);
       7: MessageDlg('Поле Edit2 не должно содержать ноль!', mtError, [mbOK], 0);
    else MessageDlg('Неизвестная ошибка.', mtInformation, [mbOK], 0);
    end
  else MessageDlg(EInstance.Message, mtError, [mbOK], 0);
end;

// Процедура, при выполнении которой возможны ошибки
procedure TForm1.btnDivideClick(Sender: TObject);
var x, y: integer;
     res: real;
```

```
begin
  // Преобразование содержимого полей редактирования в целые числа
  try
    x := StrToInt(Edit1.Text);
    y := StrToInt(Edit2.Text);
  except
    on EO: EConvertError do
      Raise EExceptionCode.Create(EO.Message, 19)
    else Raise EExceptionCode.Create('Неизвестная ошибка.', 0);
  end;
  // Деление целых чисел
  try
    res := x / y;
  except
    on EO: EZeroDivide do
       Raise EExceptionCode.Create(EO.Message, 7)
    else Raise EExceptionCode.Create('Неизвестная ошибка.', 0);
  end;
  // Вывод результата
  Edit3.Text := FloatToStr(res);
end;
```

В приведенном примере создается, вызывается и обрабатывается новое исключение класса `EExceptionCode`. Новый класс, в отличие от базового класса `Exception`, содержит не только описание возникающей ошибки (в свойстве `Message`), но и код ошибки. Для этого в описание класса `EExceptionCode` введены новое поле `FErrCode` и новое свойство `ErrorCode`, а также переопределен конструктор `Create`.

Обработка исключений выполняется с помощью глобального обработчика — процедуры `ProcExcept` в составе специального класса `GlobalHandler`. Если исключение принадлежит новому классу `EExceptionCode`, то проверяется его код `ErrCode` и выдаются соответствующие предупреждения; если же исключение принадлежит другому классу, то выдается текст из свойства `Message`, поясняющий ошибку. (На практике процесс обработки ошибок обычно более сложен и разветвлен.)

Часть действий, связанных с определением глобального обработчика, выполнена непосредственно в модуле формы `Form1`. В файле проекта для события `OnException` требуется указать новый глобальный обработчик:

```
Application.OnException := GlobalHandler.ProcExcept;
```

Исключения нового класса `EExceptionCode` могут генерироваться в процедуре, выполняемой при нажатии кнопки `btnDivide`. В этой процедуре производится деление двух целых чисел, введенных в поля редактирования `Edit1` и `Edit2`. Как наиболее вероятные проверяются два вида исключений: ошибка преобразования и деление на ноль. Первой из этих ошибок соответствует код 19, второй — код 7. Если возникла другая ошибка, то полю присваивается значение, равное нулю. Значения кодов устанавливаются программно с учетом назначения приложения и обработки возникающих ошибок.

Особенности отладки обработчиков исключений

Процесс отладки обработчиков исключений в интегрированной среде разработки Delphi имеет некоторые особенности.

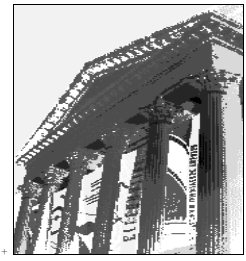
По умолчанию при возникновении динамической ошибки Delphi перехватывает исключение, выдает соответствующее сообщение и отображает в тексте модуля строку кода, вызвавшую исключение. После этого можно прервать выполнение приложения, нажав комбинацию клавиш <Ctrl>+<F2> или выполнив команду **Run | Program reset**. В этом случае процедуры приложения, предназначенные для обработки исключений, не выполняются.

Если после выдачи сообщения от встроенного отладчика Delphi требуется продолжить работу приложения, то необходимо выполнить соответствующую команду меню, например **Run | Run**, или нажать клавишу <F9>. В данном случае выполнение программы возобновляется и возникшее исключение обрабатывается средствами приложения.

Чтобы увидеть работу обработчиков исключений в "чистом виде", т. е. без вмешательства отладчика Delphi, можно запустить приложение непосредственно из Windows, например, с помощью Проводника. Другим способом является запрещение работы встроенного отладчика. В этом случае программа не будет останавливаться в случае возникновения исключений, однако станут недоступными и отладочные команды.

Поэтому рекомендуется отключать отладчик только на время тестирования программных средств обработки исключений. Для отключения отладчика следует сбросить флажок **Integrated debugging** (Интегрированная отладка) на странице **General** (Общие) окна **Debugging Options** (Параметры отладки). Окно параметров отладки вызывается командой **Tools | Debugger Options** (Сервис | Параметры отладки).

ГЛАВА 8



Сложные элементы интерфейса

При изучении визуальных компонентов нами рассмотрены наиболее простые элементы управления, предназначенные, например, для отображения текста (компонент `Label`) или однострочного редактирования данных (компонент `Edit`). В этой главе описываются более сложные элементы пользовательского интерфейса: полоса прокрутки, ползунок, счетчик, строка состояния, таблица и блокнот.

Работа с диапазоном значений

Работа с диапазоном значений заключается в выборе и задании целочисленных значений с помощью ползунка. В Delphi для этого можно использовать компоненты `ScrollBar` (полоса прокрутки) и `TrackBar` (ползунок), расположенные соответственно на страницах **Standard** и **Win32** Палитры компонентов.

Оба элемента управления (рис. 8.1) представляют собой вертикально или горизонтально ориентированную полосу с ползунком. Ползунок можно передвигать с помощью мыши или клавиш управления курсором, а также клавиш <Page Up> и <Page Down>.

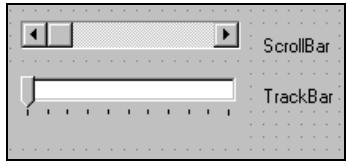


Рис. 8.1. Компоненты `ScrollBar` и `TrackBar`

Обычно компонент `ScrollBar` применяется для прокрутки информации, поэтому его называют *полосой прокрутки*, или *скроллером*. Ползунок полосы прокрутки указывает относительное расстояние, на которое отображаемый фрагмент информации отстоит от ее краев. Размер ползунка может изменяться для указания видимой в настоящий момент доли информации. Управление полосой прокрутки осуществляется перетаскиванием ползунка с помощью мыши или щелчком мыши на стрелках в начале и конце полосы.

Компонент `TrackBar` называется *ползунком*, или *шкалой*, и обычно используется для изменения значений в заданном диапазоне или выбора целых чисел внутри диапазона. В Windows, например, ползунок применяется при задании размеров Корзины, в регуляторе громкости звука (рис. 8.2), при выборе разрешения монитора и т. д. Кроме полосы, по которой он передвигается, ползунок содержит риски для отсчета значений управляемого параметра.

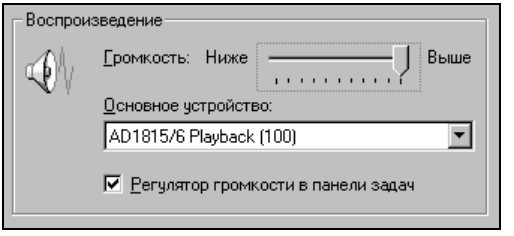


Рис. 8.2. Регулятор громкости звука

Хотя компоненты `ScrollBar` и `TrackBar` обычно используются для разных целей, происходят они от одного класса `TWinControl` и имеют схожее поведение и характеристики.

Для обоих компонентов свойства `Min` и `Max` типа `Integer` задают диапазон изменения возможных значений, а свойство `Position` типа `Integer` определяет текущую позицию ползунка в диапазоне. Пользователь изменяет значение свойства `Position`, перемещая ползунок. Допускается изменять любое из этих свойств и программно, устанавливая для них требуемые значения при разработке или при выполнении приложения. Для компонента `ScrollBar` дополнительно можно использовать метод `SetParams(APosition, AMax, AMin: Integer)`, позволяющий установить значения всех указанных свойств.

Замечание

Ни визуально, ни программно нельзя установить текущую позицию (значение свойства `Position`), выходящую за рамки допустимого диапазона (значения свойств `Min` и `Max`).

Расположение полосы компонента `ScrollBar` по горизонтали или вертикали определяет свойство `Kind` типа `TScrollBarKind`, принимающее значения:

- `sbHorizontal` (по горизонтали);
- `sbVertical` (по вертикали).

Для компонента `TrackBar` с той же целью используется свойство `Orientation` типа `TTrackBarOrientation`, принимающее значения:

- `trHorizontal` (по горизонтали);
- `trVertical` (по вертикали).

Свойства компонента `ScrollBar SmallChange` и `LargeChange` (типа `TScrollBarInc`) и свойства компонента `TrackBar LineSize` и `PageSize` (типа `Integer`) определяют *шаг перемещения* ползунка при управлении с клавиатуры. Свойства, указанные первыми, определяют шаг перемещения ползунка при использовании клавиш управления курсором, а свойства, указанные вторыми, задают шаг перемещения при использовании клавиш <Pg Up> и <Pg Down>. По умолчанию значения всех указанных свойств равны единице.

При визуальном или программном изменении позиции ползунка возникает событие `OnChange` типа `TNotifyEvent`. В случае визуального перемещения ползунка для компонента `ScrollBar` перед событием `OnChange` дополнительно генерируется событие `OnScroll` типа `TScrollEvent`. Тип `TScrollEvent` объявлен так:

```
type TScrollEvent = procedure(Sender: TObject; ScrollCode: TScrollCode;
                              var ScrollPos: Integer) of object;
```

Параметр `ScrollPos` определяет *позицию ползунка*, а параметр `ScrollCode` содержит *код состояния* полосы прокрутки и может принимать следующие значения:

- `scLineUp` — нажата клавиша <↑> или <←> или сделан щелчок мышью на верхней (левой) стрелке полосы прокрутки;
- `scLineDown` — нажата клавиша <↓> или <→> или сделан щелчок мышью на нижней (правой) стрелке полосы прокрутки;
- `scPageUp` — нажата клавиша <Page Up> или сделан щелчок мышью сверху (слева) от ползунка;
- `scPageDown` — нажата клавиша <Page Down> или сделан щелчок мышью снизу (справа) от ползунка;
- `scPosition` — ползунок перемещен мышью и отпущен;
- `scTrack` — в настоящий момент выполняется перетаскивание ползунка;
- `scTop` — ползунок перемещен мышью в крайнюю верхнюю (левую) позицию;
- `scBottom` — ползунок перемещен мышью в крайнюю нижнюю (правую) позицию;
- `scEndScroll` — перетаскивание ползунка закончено.

События `OnChange` и `OnScroll` можно использовать для программного управления операциями, связанными с прокруткой информации.

Замечание

Событие `OnScroll` генерируется только при перемещении ползунка пользователем.

Рассмотрим пример использования полосы прокрутки.

Полоса прокрутки `ScrollBar1` управляет горизонтальным положением надписи `Label1` в форме `Form4` (рис. 8.3). Расположенная над полосой прокрутки надпись `lblStatus` отражает состояние операции перемещения.

В листинге 8.1 приведен текст модуля `uScBar` с описанием главной формы `Form4`.

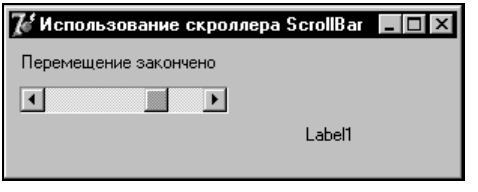


Рис. 8.3. Использование полосы прокрутки

Листинг 8.1. Пример использования полосы прокрутки

```
unit uScBar;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ComCtrls, StdCtrls;
type
  TForm4 = class(TForm)
    ScrollBar1: TScrollBar;
     lblStatus: TLabel;
        Label1: TLabel;

    procedure FormCreate(Sender: TObject);
    procedure ScrollBar1Scroll(Sender: TObject; ScrollCode: TScrollCode;
      var ScrollPos: Integer);
    procedure FormResize(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form4: TForm4;
implementation
{$R *.DFM}

procedure TForm4.FormCreate(Sender: TObject);
begin
  lblStatus.Caption := 'Начальное положение';
end;

procedure TForm4.FormResize(Sender: TObject);
begin
  // Установка параметров полосы прокрутки
  with ScrollBar1 do begin
     Min := 0;
     Max := Form4.ClientWidth - Label1.Width;
     Position := Max div 2;
     SmallChange := 1;
     LargeChange := Max div 10;
  end;
  // Помещение надписи Label1 в начальную позицию (центр окна)
  Label1.Left := ScrollBar1.Position;
end;

procedure TForm4.ScrollBar1Scroll(Sender: TObject; ScrollCode: TScrollCode;
  var ScrollPos: Integer);
begin
  // Перемещение надписи Label1 соответственно позиции ползунка
  Label1.Left := ScrollPos;
```

```
  // Вывод информации о состоянии полосы прокрутки
  case ScrollCode of
        scTrack: lblStatus.Caption := 'Ползунок перемещается';
    scEndScroll: lblStatus.Caption := 'Перемещение закончено';
  end;
end;
end.
```

Перемещение надписи выполнено в обработчике события `OnScroll` полосы прокрутки. При использовании для этих целей события `OnChange` его обработчик содержит схожий код, при этом позиция ползунка управляется с помощью свойства `Position`. Кроме того, в обработчике события `OnChange` не анализируется состояние операции перемещения, т. к. отсутствует параметр `ScrollCode`.

Чтобы программа могла правильно перемещать надпись `Label1` при изменении размеров окна, установка параметров полосы прокрутки выполняется при генерации события `OnResize` формы.

Ползунок `TrackBar` может содержать на шкале риски (метки), поэтому у него есть соответствующие свойства для управления их размещением. *Стиль рисок* задается свойством `TickStyle` типа `TTickStyle`, принимающим следующие значения:

- `tsAuto` (автоматическая расстановка рисок) — по умолчанию. Частота рисок задается свойством `Frequency` типа `Integer`, его значение определяет число из диапазона `Min...Max`, которому соответствует одна метка;
- `tsManual` (риски отображаются на концах шкалы). Программно можно установить риску в любой позиции с помощью метода `SetTick(Value: Integer)`, параметр которого задает местоположение метки;
- `tsNone` (риски отсутствуют).

Риски на шкале могут находиться в различных *позициях*, определяемых свойством `TickMarks` типа `TTickMark`, которое может принимать следующие значения:

- `tmBottomRight` (риски располагаются внизу горизонтальной шкалы и справа вертикальной шкалы);
- `tmTopLeft` (риски располагаются сверху горизонтальной шкалы и слева вертикальной шкалы);
- `tmBoth` (риски располагаются по обе стороны шкалы).

Программно можно выделить внутри шкалы компонента `TrackBar` произвольный диапазон, визуально выделяемый системным цветом. Границы выделенной области определяются свойствами `SelStart` и `SelEnd` типа `Integer`, а на шкале отмечаются треугольными рисками.

Рассмотрим пример использования компонента `TrackBar` для управления размером и цветом компонента `Panel1`, расположенного в правой части окна (рис. 8.4).

Ползунок `tbSize` изменяет длину стороны квадратной панели от двадцати до ста пикселов. Назначение и диапазон ползунка поясняются надписями.

В листинге 8.2 приведен текст модуля `uTrBar`, реализующего главную форму приложения.

Рис. 8.4. Управление цветом и размером панели

Листинг 8.2. Пример использования компонента `TrackBar`

```
unit uTrBar;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, ComCtrls, StdCtrls;
type
  TForm6 = class(TForm)
     tbColorR: TTrackBar;
       tbSize: TTrackBar;
       Panel1: TPanel;
       Label2: TLabel;
       Label3: TLabel;
       Label4: TLabel;
       Label5: TLabel;
       Label6: TLabel;
       Label7: TLabel;
     tbColorB: TTrackBar;
     tbColorG: TTrackBar;
    lblColorG: TLabel;
    lblColorB: TLabel;
    lblColorR: TLabel;

    procedure FormCreate(Sender: TObject);
    procedure tbSizeChange(Sender: TObject);
    procedure tbColorChange(Sender: TObject);
    procedure Panel1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form6: TForm6;
```

```
implementation
{$R *.DFM}

procedure TForm6.FormCreate(Sender: TObject);
begin
  // Установка параметров ползунка, управляющего размером панели
  with tbSize do begin
     Orientation := trVertical;
     Min := 20;
     Max := 100;
     Position := 100;
     Frequency := Max div 10;
     LineSize := 1;
     PageSize := Max div 10;
     OnChange := tbSizeChange;
  end;
  tbSizeChange(Sender);
  // Установка параметров ползунка, управляющего красным цветом панели
  with tbColorR do begin
     Orientation := trVertical;
     Min := 0;
     Max := 255;
     Position := Max div 2;
     Frequency := Max div 20;
     LineSize := Max div 20;
     PageSize := Max div 20;
     OnChange := tbColorChange;
  end;
  // Установка параметров ползунка, управляющего зеленым цветом панели
  with tbColorG do begin
     Orientation  := trVertical;
     Min := 0;
     Max := 255;
     Position := Max div 2;
     Frequency := Max div 20;
     LineSize := Max div 20;
     PageSize := Max div 20;
     OnChange := tbColorChange;
  end;
  // Установка параметров ползунка, управляющего синим цветом панели
  with tbColorB do begin
     Orientation := trVertical;
     Min := 0;
     Max := 255;
     Position := Max div 2;
     Frequency := Max div 20;
     LineSize := Max div 20;
     PageSize := Max div 20;
     OnChange := tbColorChange;
  end;
```

```
  tbColorChange(Sender);
  lblColorR.Font.Color := clRed;
  lblColorG.Font.Color := clGreen;
  lblColorB.Font.Color := clBlue;
end;

procedure TForm6.tbSizeChange(Sender: TObject);
begin
  Panel1.Height := tbSize.Position;
  Panel1.Width := tbSize.Position;
  Panel1.Left := Form6.ClientWidth - 60 - Panel1.Width div 2;
  Panel1.Top := Form6.ClientHeight div 2 - Panel1.Height div 2;
end;

// Общий обработчик для всех ползунков, управляющих цветом панели
procedure TForm6.tbColorChange(Sender: TObject);
begin
  Panel1.Color := RGB(tbColorR.Position, tbColorG.Position, tbColorB.Position);
end;

procedure TForm6.Panel1Click(Sender: TObject);
begin
  Close;
end;
end.
```

Для изменения цвета используются три ползунка: `tbColorR`, `tbColorG` и `tbColorB`, каждый из которых управляет своей составляющей цвета: красной, зеленой и синей. Результирующий цвет панели образуется как сумма трех составляющих. Назначение всех ползунков отражается в расположенных рядом с ними надписях. Для ползунков, управляющих цветом панели, используется общий обработчик `tbColorChange` события `OnChange`. Преобразование трех составляющих в результирующий цвет осуществляется посредством API-функции `RGB`. Закрытие окна и завершение работы приложения осуществляется щелчком мыши на панели.

Реверсивные счетчики

Реверсивный счет — это увеличение или уменьшение числового значения на заданную величину (приращение). Для организации такого счета в Delphi предназначены компоненты `UpDown` (расположен на странице **Win32** Палитры), `SpinButton` и `SpinEdit` (расположены на странице **Samples**), которые похожи и имеют много общих характеристик. Все эти компоненты являются *счетчиками* и имеют две кнопки со стрелками, используемыми для увеличения или уменьшения значения счетчика. На рис. 8.5 показана группа **Общие параметры настройки** окна параметров текстового процессора Microsoft Word, в которой счетчик используется для управления длиной списка запоминаемых файлов.

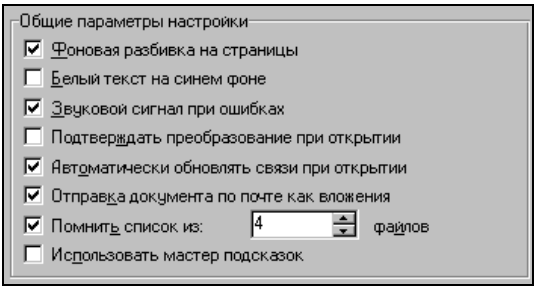


Рис. 8.5. Использование счетчика для задания длины списка файлов

Компонент *UpDown*

Счетчик `UpDown` существует как автономный элемент управления и не имеет поля, в котором отображается изменяемое число; однако обычно он связан с другим оконным элементом управления, чаще всего с однострочным редактором `Edit` (рис. 8.6) или статическим текстом `StaticText`. Использовать для этих целей надпись `Label` нельзя, т. к. она не является оконным элементом управления. Вместо редактора `Edit` можно использовать другие компоненты, например, многострочные редакторы `Memo` и `RichEdit`, но на практике это применяется редко.

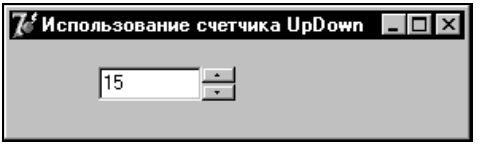


Рис. 8.6. Счетчик, ассоциированный с однострочным редактором

Счетчик и ассоциированный с ним элемент взаимно дополняют друг друга и образуют парный реверсивный элемент управления: поле содержит число, а счетчик служит для уменьшения или увеличения значения числа. Счетчик `UpDown` автоматически располагается рядом со своим компонентом-"партнером". Реверсивный элемент управления не имеет заголовка, поэтому, если нужно пояснить назначение счетчика, возле него располагается надпись.

Подключение счетчика `UpDown` к ассоциированному компоненту выполняется с помощью свойства `Associate` типа `TWinControl`. В качестве парного для счетчика можно указать любой оконный элемент управления, например, `Button`, `CheckBox` или `StatusBar`. Обычно со счетчиком связывают компоненты редактирования. Сразу после установления связи между двумя компонентами счетчик выравнивается относительно указанного элемента управления и размещается в его поле. Например, связать счетчик с элементом редактирования можно так:

```
UpDown1.Associate := Edit1;
```

На практике такая связь обычно устанавливается при проектировании с помощью Инспектора объектов.

Счетчик может выравниваться в поле ассоциированного компонента различными способами. *Вариант выравнивания* определяет свойство `AlignButton` типа `TUDAlignButton`, принимающее следующие значения:

- `udLeft` (по левому краю);
- `udRight` (по правому краю) — по умолчанию.

Вид счетчика определяется свойством `Orientation` типа `TUDOrientation`, принимающим следующие значения:

- `udHorizontal` (стрелки счетчика направлены вправо и влево);
- `udVertical` (стрелки счетчика направлены вверх и вниз) — по умолчанию.

Замечание

Даже в случае горизонтальной ориентации стрелок (`udHorizontal`) при управлении с клавиатуры счетчик реагирует только на нажатия клавиш <↑> и <↓>.

Числовой диапазон значения, управляемого реверсивным компонентом, задается свойствами `Min` и `Max` типа `SmallInt`. Эти свойства задают минимальное/максимальное возможные значения соответственно.

Шаг изменения значения реверсивного элемента управления при нажатии кнопки со стрелкой содержится в свойстве `Increment` типа `Integer`. По умолчанию значение этого свойства равно единице. Его можно изменять программно, устанавливая нужное целое число.

Кроме мыши, можно управлять счетчиком с помощью клавиатуры. Возможность использования *клавиатуры* зависит от свойства `ArrowKeys` типа `Boolean`. По умолчанию оно имеет значение `True`, и при нахождении реверсивного элемента в фокусе ввода значение счетчика можно изменять с помощью клавиш управления курсором (<↑> и <↓>). Если свойство `ArrowKeys` имеет значение `False`, то нажатие клавиш управления курсором не влияет на счетчик.

Текущую позицию реверсивного элемента управления определяет значение свойства `Position` типа `SmallInt`. Это значение должно находиться в диапазоне, определяемом свойствами `Min` и `Max`, независимо от того, программно или визуально оно изменяется. Когда значение свойства `Position` выходит на границу допустимого диапазона, оно фиксируется или обращается в противоположное, что зависит от свойства `Wrap` типа `Boolean`. Если это свойство установлено в `True`, то, например, при попытке пользователя увеличить текущую позицию счетчика свыше значения `Max` свойство `Position` принимает значение `Min`. По умолчанию свойство `Wrap` имеет значение `False`, и значение свойства `Position` фиксируется при достижении границы (значения `Max` или `Min`).

Значение реверсивного счетчика содержится также в текстовом свойстве ассоциированного элемента управления, например, в свойстве `Text` компонента `Edit`. Управлять текущим значением реверсивного счетчика можно с помощью обоих свойств `Position` и `Text`, результат будет одинаков. Если значение поля участвует в *вычислениях*, то более предпочтительным является свойство `Position`, т. к. в этом случае не нужно преобразовывать строковое значение в числовое и обратно. В случае, когда значение счетчика используется как *строка*, удобнее применять свойство `Text`.

Пример использования реверсивного счетчика:

```
UpDown1.Associate := Edit1;
...
Label1.Caption := Edit1.Text;
TabControl1.TabIndex := UpDown1.Position;
```

Здесь надпись `Label1` отображает значение реверсивного счетчика как строку, а элемент `TabControl1` для активизации вкладки использует значение реверсивного счетчика `UpDown1` как число.

В случаях, когда компонент `UpDown` связан с элементом редактирования, пользователь может изменять значение счетчика не только с помощью кнопок со стрелками, но и прямым редактированием с клавиатуры.

Замечание

Если ввести в поле редактора данные, которые не являются целым числом, принадлежащим заданному диапазону, то исключение не генерируется. Однако значения свойства `Position` счетчика и свойства `Text` ассоциированного с ним редактора не будут соответствовать друг другу. Свойство `Text` будет содержать значение, введенное пользователем, а свойство `Position` сохранит свое предыдущее значение. При нажатии кнопки со стрелкой счетчика информация в поле ассоциированного компонента будет приведена в соответствие со значением свойства `Position`.

Можно запретить непосредственное редактирование в поле парного к счетчику компонента программно, установив его свойство `ReadOnly` в значение `True`.

В текстовом представлении текущего значения счетчика (т. е. значения его свойства `Position`), которое отображается в ассоциированном компоненте, в качестве разделителя тысяч может использоваться запятая. Запятая появляется, если свойство `Thousands` типа `Boolean` установлено в значение `True` (по умолчанию), в противном случае запятая отсутствует.

При нажатии пользователем кнопок счетчика возникают события `OnChanging` и `OnClick`.

Событие `OnChanging` типа `TUDChangingEvent` генерируется при попытке *изменения значения счетчика*. Тип этого события описан следующим образом:

```
type TUDChangingEvent = procedure(Sender: TObject; var AllowChange:
  Boolean) of object;
```

Параметр `AllowChange` позволяет принять или запретить изменение текущей позиции счетчика. Если по какой-либо причине изменение текущей позиции нежелательно, то этот параметр необходимо установить в значение `False`.

В случаях, когда требуется проанализировать, какая из двух кнопок счетчика была нажата, можно использовать событие `OnClick` типа `TUDClickEvent`, описанного как:

```
type TUDClickEvent = procedure(Sender: TObject; Button: TUDBtnType)
  of object;
```

Параметр `Button` имеет следующие допустимые значения:

- `btNext` (нажата кнопка увеличения значения счетчика);
- `btPrev` (нажата кнопка уменьшения значения счетчика).

Компонент *SpinButton*

Счетчик SpinButton представляет собой две кнопки со стрелками. Его также можно использовать для реверсивного счета, но по сравнению со счетчиком UpWown он более прост и не содержит указателя текущей позиции Position и таких свойств, как Min, Max или Increment. При применении компонента SpinButton программист сам управляет размещением числа и изменением его значения. Обычно для хранения числа объявляется глобальная переменная, а изменение ее значения выполняют обработчики событий нажатия кнопок. При необходимости значение переменной можно отобразить с помощью какого-либо визуального компонента, скажем, надписи Label, а также установить значение этой переменной, например, с помощью редактора Edit.

При нажатии кнопок компонента SpinButton генерируется не одно общее событие, например, OnChanging, как в предыдущем случае, а по одному отдельному событию для каждой из кнопок — OnUpClick и OnDownClick типа TNotifyEvent.

Рассмотрим пример использования компонента SpinButton:

```
var Number: longint;
      Step: Integer;
...
procedure TForm1.FormCreate(Sender: TObject);
begin
  Number := 20;
  Step := 5;
  Label1.Caption := IntToStr(Number);
end;

procedure TForm1.SpinButton1DownClick(Sender: TObject);
begin
  Number := Number - Step;
  Label1.Caption := IntToStr(Number);
end;

procedure TForm1.SpinButton1UpClick(Sender: TObject);
begin
  Number := Number + Step;
  Label1.Caption := IntToStr(Number);
end;
```

Нажатие кнопок компонента SpinButton1 приводит к изменению значения переменной Number на величину Step. Число Number отображается в форме с помощью надписи Label1.

Рисунки, выводимые на кнопках компонента SpinButton (по умолчанию — изображения стрелок), можно изменять программно. Рисунки кнопок определяются свойствами UpGlyph и DownGlyph типа TBitmap, значения которых обычно устанавливаются при разработке приложения.

Компонент *SpinEdit*

Счетчик SpinEdit по своему внешнему виду и функциональным возможностям соединяет в себе счетчик UpDown и ассоциированный с ним редактор Edit. Соответственно,

поведение компонента SpinEdit и его характеристики являются аналогами поведения и характеристик счетчика UpDown, рассмотренного ранее. Наиболее характерными для элемента SpinEdit являются свойства Value, MinValue, MaxValue, Increment типа Integer, ReadOnly типа Boolean, событие OnChange типа TNotifyEvent.

Строка состояния

Строка состояния, или *строка статуса*, представляет собой элемент управления, который отображает текущую информацию о состоянии содержимого окна и клавиатуры, контекстные подсказки по текущему пункту меню или кнопке панели инструментов и другие сведения. Строка состояния, как правило, выровнена по нижнему краю главного окна приложения. Для работы со строками состояний в Delphi имеется специальный компонент StatusBar, находящийся в Палитре компонентов на странице **Win32**. Кроме того, строку состояния можно создать на базе компонента Panel.

Создание строки состояния

Строку состояния можно создать программно, разместив для этого в форме компонент-панель (Panel). Обычно эта панель не имеет заголовка и выровнена по нижнему краю формы.

Внутри строки состояния размещается несколько дополнительных панелей, служащих для отображения информации. Число таких вложенных панелей зависит от объема отображаемых сведений. Вместо панелей для вывода текущей информации можно использовать отдельный компонент, чаще всего Label.

Рассмотрим создание строки состояния на примере. Наша строка состояния содержит три информационных поля (рис. 8.7), отображающих текущие дату, время и координаты указателя мыши.

Рис. 8.7. Вид строки состояния

В листинге 8.3 приводится код модуля uStatus для формы Form1, в окне которой содержится строка состояния.

Листинг 8.3. Пример строки состояния

```
unit uStatus;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls;
```

```
type
  TForm1 = class(TForm)
    StatusPanel: TPanel;
     InfoPanel1: TPanel;
     InfoPanel2: TPanel;
     InfoPanel3: TPanel;

    procedure IdleProc(Sender :TObject; var Done :Boolean);
    procedure FormResize(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnIdle := IdleProc;
  // Установить значения этих свойств можно
  // с помощью Инспектора объектов
  StatusPanel.Align := alBottom;
  InfoPanel1.BevelInner := bvNone;
  StatusPanel.BevelOuter := bvRaised;
  StatusPanel.Caption := '';
  InfoPanel1.Align := alNone;
  InfoPanel2.Align := alNone;
  InfoPanel3.Align := alNone;
  InfoPanel1.Alignment := taCenter;
  InfoPanel2.Alignment := taCenter;
  InfoPanel3.Alignment := taCenter;
  InfoPanel1.BevelInner := bvNone;
  InfoPanel2.BevelInner := bvNone;
  InfoPanel3.BevelInner := bvNone;
  InfoPanel1.BevelOuter := bvLowered;
  InfoPanel3.BevelOuter := bvLowered;
  InfoPanel3.BevelOuter := bvLowered;
  InfoPanel3.Caption := ' ';
end;

procedure TForm1.FormResize(Sender: TObject);
var pnlLength: integer;
begin
  pnlLength := Form1.ClientWidth div 3;
  InfoPanel1.Width := pnlLength;
```

```
  InfoPanel2.Width := pnlLength;
  InfoPanel2.Width := pnlLength;
  InfoPanel1.Left := PanelStatus.Left;
  InfoPanel2.Left := PanelInfo1.Left + pnlLength;
  InfoPanel3.Left := PanelInfo2.Left + pnlLength;
end;

procedure TForm1.IdleProc(Sender :TObject; var Done :Boolean);
begin
  InfoPanel1.Caption := DateToStr(Date);
  InfoPanel2.Caption := TimeToStr(Time);
  // Это присваивание нужно для того, чтобы в строке состояния
  // информация об изменении даты и времени обновлялась немедленно
  Done := False;
end;

procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
                               X, Y: Integer);
begin
  InfoPanel3.Caption := IntToStr(X) + '  ' + IntToStr(Y);
end;
end.
```

Для построения строки состояния использованы компонент-панель `StatusPanel` и три вложенных компонента панели `InfoPanel1`, `InfoPanel2` и `InfoPanel3`, отображающие соответствующую информацию. Текущие координаты указателя мыши при нахождении его в форме `Form1` отслеживаются и отображаются с помощью процедуры `FormMouseMove`, вызываемой каждый раз, когда мышь перемещается.

Данные о дате и времени постоянно обновляются в процедуре `IdleProc`, которая для приложения является обработчиком события `OnIdle`. Для того чтобы сделанные в этой процедуре изменения отображались в строке состояния немедленно, параметру `Done` присваивается значение `False`, что предписывает системе обработать информацию, не дожидаясь других сообщений. По умолчанию параметр `Done` имеет значение `True`, и отображение времени будет обновляться только в случае прихода новых сообщений, например, при перемещении в форме указателя мыши.

Главная панель строки состояния выровнена по нижнему краю формы и при изменении размеров окна будет автоматически изменять свои размеры. Для информационных панелей пересчет их ширины и выравнивание производятся в процедуре `FormResize`.

При создании формы `Form1` в процедуре `FormCreate` определяется обработчик события `OnIdle` приложения, а также задаются основные свойства для всех четырех панелей. Установить значения свойств панелей можно и с помощью Инспектора объектов при проектировании приложения.

Компонент *StatusBar*

Удобным способом отображения информации о состоянии является использование специального компонента `StatusBar`, находящегося в Палитре компонентов на страни-

це **Win32**. Компонент `StatusBar` представляет собой *строку состояния*, которая может иметь одну или несколько панелей для вывода текстовой информации.

Возможность поддержки строкой состояния *нескольких панелей* определяет свойство `SimplePanel` типа `Boolean`. Если свойство имеет значение `True`, то в строке состояния имеется только одна панель, и выводимый на ней текст можно задать через свойство `SimpleText` типа `String`. Если свойство `SimplePanel` имеет значение `False` (по умолчанию), то в строке состояния имеется несколько панелей, и текст каждой из них устанавливается с помощью свойства `Text` типа `String` отдельной панели.

Для создания нескольких панелей и управления ими предназначено свойство `Panels` типа `TStatusPanels`, позволяющее обращаться к панелям строки состояния как к элементам массива, нумерация которых начинается с нуля (соответственно, первый элемент — `Panels[0]`, второй — `Panels[1]` и т. д.).

Основные свойства панели строки состояния:

- `Alignment` типа `Talignment` — задает способ выравнивания текста относительно панели и принимает следующие значения:
  - `aLeftJustify` (по левому краю) — по умолчанию;
  - `taCenter` (по центру);
  - `taRightJustify` (по правому краю);
- `Bevel` типа `TStatusPanelBevel` — определяет вид панели относительно поверхности строки состояния и принимает следующие значения:
  - `pbNone` (плоская);
  - `pbLowered` (углубленная) — по умолчанию;
  - `pbRaised` (приподнятая);
- `Style` типа `TStatusPanelStyle` — определяет способ отображения на панели информации и принимает следующие значения:
  - `psText` (выводится текст из свойства `Text`) — по умолчанию;
  - `psOwnerDraw` (программно выводится текстовая и/или графическая информация);
- `Text` типа `String` — содержит текст, отображаемый на поверхности панели;
- `Width` типа `Integer` — задает ширину панели; по умолчанию все панели, кроме последней, имеют ширину 50 пикселов, а последняя панель занимает все свободное место строки состояния.

Пример использования панели строки состояния:

```
StatusBar1.Panels[1].Text := 'Время ' + TimeToStr(Time);
```

Здесь информация о текущем времени отображается во второй панели строки состояния `StatusBar1`.

Панели строки состояния можно настраивать как во время разработки, так и в процессе выполнения приложения.

Для настройки панелей на этапе разработки приложения используется редактор панелей строки состояния (рис. 8.8), который можно активизировать двойным щелчком в

окне Инспектора объектов на значении свойства `Panels` компонента `StatusBar`. Управление отдельными панелями строки состояния осуществляется путем установки их свойств (видимых в Инспекторе объектов после вызова редактора панелей) в нужные значения.

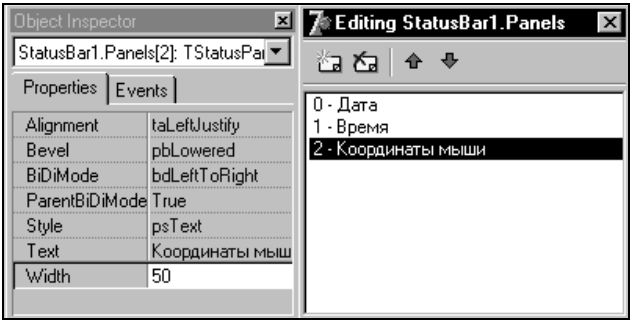


Рис. 8.8. Редактор панелей строки состояния

При управлении панелями строки состояния в процессе выполнения приложения кроме свойств, видимых в Инспекторе объектов, можно дополнительно использовать перечисленные далее свойства и методы.

Свойство `Count` типа `Integer` доступно только для чтения и показывает *число панелей* в строке состояния.

Метод `Add` позволяет динамически *добавлять новую панель* к строке состояния. После добавления панели ее свойства принимают значения по умолчанию, новые значения (если они необходимы) требуется устанавливать программно.

Например, в процедуре

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  StatusBar1.Panels.Add;
  StatusBar1.Panels[StatusBar1.Panels.Count – 1].Text := 'Новая панель';
end;
```

при нажатии кнопки `Button2` к строке состояния `StatusBar1` формы `Form2` добавляется новая панель с текстом `Новая панель`.

Если форма, в которой находится строка состояния, допускает изменение размеров, то в правом конце строки состояния может находиться *масштабирующий захват*, облегчающий уменьшение или увеличение ее размеров. Отображением захвата управляет свойство `SizeGrip` типа `Boolean`. Если свойство `SizeGrip` имеет значение `True`, то масштабирующий захват есть, в противном случае — нет.

Замечание

Если строка состояния выровнена так, что масштабирующий захват находится в правом нижнем углу формы, то использование захвата приводит к изменению размеров самой формы, а не строки состояния.

Рассмотрим пример использования строки состояния. В строке состояния `StatusBar1`, расположенной в форме `Form1`, отображаются текущее время и информация о состоя-

нии клавиш <Num Lock> и <Insert>. В строке состояния присутствует масштабируемый захват (рис. 8.9).

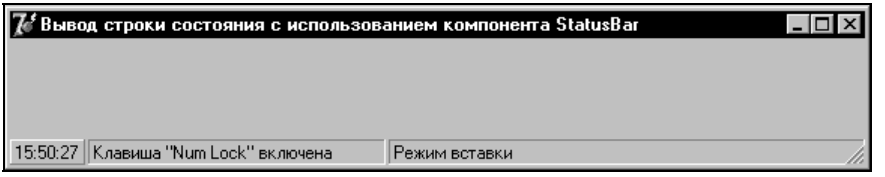


Рис. 8.9. Вывод строки состояния с помощью компонента `StatusBar`

Анализ и отображение информации на панелях строки состояния обеспечивает процедура `IdleProc`, которая является обработчиком события `OnIdle` приложения. Текст этой процедуры приводится далее.

```
procedure TForm1.IdleProc(Sender :TObject; var Done :Boolean);
begin
  StatusBar1.Panels[0].Text := TimeToStr(Time);
  if GetKeyState(VK_NumLock) = 1
     then StatusBar1.Panels[1].Text := 'Клавиша "Num Lock" выключена'
     else StatusBar1.Panels[1].Text := 'Клавиша "Num Lock" включена';
  if GetKeyState(VK_Insert) = 1
     then StatusBar1.Panels[2].Text := 'Режим замены'
     else StatusBar1.Panels[2].Text := 'Режим вставки';
  Done := False;
end;
```

Для проверки состояния клавиш используется API-функция `GetKeyState`. Функция `GetKeyState(nVirtKey: int): Short` получает в качестве параметра виртуальный код `nVirtKey` клавиши и как результат возвращает целое число с признаками состояния этой клавиши. Если старший бит равен единице, то клавиша нажата, если равен нулю — отпущена. Для клавиш-переключателей типа <Num Lock> при включенном переключателе функция возвращает в младшем бите единицу, а при выключенном — ноль. Для задания виртуальных кодов клавиш удобно использовать константы с именами вида `VK_xxx`, например, `VK_Space`, `VK_F3` и `VK_W` для клавиш <Пробел>, <F3> и <W> соответственно. Для получения виртуальных кодов различных клавиш можно вызывать API-функцию `VkKeyScan`.

Часто на практике в строке состояния отображают подсказки, поясняющие назначение того или иного элемента интерфейса. Наиболее просто это выполнить, использовав свойство `Hint` компонентов и событие `OnHint` приложения.

Рассмотрим следующий пример. Пусть подсказки о назначении элементов управления выводятся во второй панели строки состояния `StatusBar1`. Так, в окне на рис. 8.10 указатель мыши находится на кнопке **Закрыть**, и панель отображает соответствующий текст.

Далее приводится код модуля `uExtend`, реализующего описанную строку состояния.

```
unit uExtend;
interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ComCtrls, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    StatusBar1: TStatusBar;
       Button1: TButton;

    procedure FormCreate(Sender: TObject);
    procedure PanelHint(Sender: TObject);
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  // Свойству Hint интерфейсных элементов должны быть присвоены
  // соответствующие значения
  Application.OnHint := PanelHint;
  StatusBar1.Panels[0].Text := 'Помощь';
end;

procedure TForm1.PanelHint(Sender: TObject);
begin
  StatusBar1.Panels[1].Text := Application.Hint;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Close;
end;
end.
```

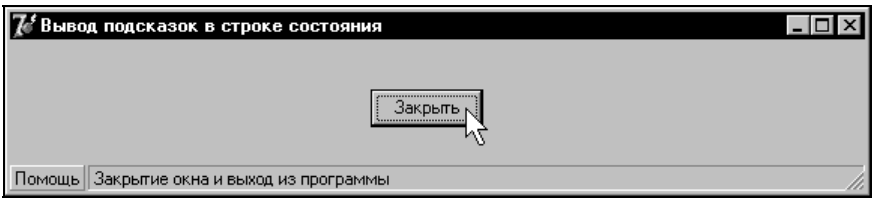


Рис. 8.10. Подсказка в строке состояния

Событию `OnHint` приложения назначается обработчик `PanelHint`. В этом обработчике выполняется вывод в строку состояния значения свойства `Hint` компонента, на котором

установлен указатель мыши. Предварительно свойствам `Hint` элементов интерфейса должны быть присвоены соответствующие значения. При этом значение свойства `ShowHint` компонентов влияет только на отображение подсказок, выводимых около самого компонента, а в строке состояния эти подсказки отображаются всегда, когда указатель мыши находится на данном элементе интерфейса.

Для обновления информации, отображаемой в строке состояния, можно применять обработчик события `OnIdle` приложения. Когда в приложении много форм, для отображения информации о состоянии удобнее использовать процедуру обработки события таймера. В этом случае в форме размещается компонент `Timer` (расположен на странице **System** Палитры компонентов) и создается обработчик его события `OnTimer`, в котором находятся инструкции, выполняющие требуемые действия по отображению текста в строке состояния. Данные в строке состояния обновляются при каждом событии таймера.

Событие таймера можно использовать и для выполнения других периодических операций.

Таблицы

Таблица представляет собой элемент, позволяющий отображать данные, разбитые на строки и столбцы. Для работы с таблицами Delphi предоставляет компоненты `DrawGrid` и `StringGrid` (таблица строк), расположенные на странице **Additional** Палитры компонентов. Эти компоненты похожи друг на друга и имеют много общего.

Таблица `DrawGrid` позволяет отображать в своих ячейках текстовую и графическую информацию. Автоматически выводится только сетка, а за прорисовку содержимого ячеек отвечает программист. Кроме того, компонент `DrawGrid` только отображает информацию, но не хранит ее. Размещение и последующее извлечение данных вне компонента `DrawGrid` также является обязанностью программиста.

Более простым и удобным для использования является компонент-таблица строк `StringGrid`, применяемый для обработки текстовых данных. Таблица строк позволяет хранить и автоматически отображать текстовую информацию. Этот компонент также называют *таблицей*, *сеткой строк* или просто *сеткой*. Несмотря на название, таблица строк способна отображать и графическую информацию. Но при этом, как и в случае с таблицей `DrawGrid`, хранение графических данных и их прорисовку программист реализует самостоятельно.

Размеры таблицы определяют свойства `ColCount` и `RowCount` типа `Longint`, задающие максимальный индекс строки и столбца соответственно. Значения этим свойствам можно присваивать и динамически — в процессе выполнения программы, что приводит к немедленному изменению размеров таблицы. По умолчанию оба свойства `ColCount` и `RowCount` имеют значение 5, что соответствует таблице размером 6×6.

Размеры ячеек в пикселах задают свойства `DefaultColWidth` и `DefaultRowHeight` типа `Integer`. Значения этих свойств действуют для всех ячеек сетки, кроме тех, для которых были установлены свои значения. Например, при добавлении нового столбца его ширина берется из свойства `DefaultColWidth`. По умолчанию свойство `DefaultColWidth` имеет значение 64, а свойство `DefaultRowHeight` — значение 24.

Так, с помощью инструкции

```
StringGrid2.ColCount := StringGrid2.ColCount + 1;
```

к таблице StringGrid2 справа добавляется новый столбец, ширина которого определяется значением свойства DefaultColWidth.

Свойства ColWidths[Index: Longint] и RowHeights[Index: Longint] типа Integer позволяют задать в пикселах ширину столбца и высоту строки с номером Index соответственно.

Если элементы не помещаются в отведенной под таблицу области, то могут автоматически появляться полосы прокрутки. Возможностью отображения полос прокрутки управляет свойство ScrollBars типа TScrollStyle, принимающее следующие значения:

- ssNone (полосы прокрутки не допускаются);
- ssHorizontal (допускается горизонтальная полоса прокрутки);
- ssVertical (допускается вертикальная полоса прокрутки);
- ssBoth (допускаются обе полосы прокрутки) — по умолчанию.

Крайние левые столбцы и верхние строки таблицы можно устанавливать фиксированными. Обычно фиксация используется для оформления заголовков. *Число фиксированных столбцов и строк* таблицы определяют свойства FixedCols и FixedRows типа Integer соответственно. По умолчанию свойства имеют значение 1. Фиксированные элементы могут выделяться цветом и при прокрутке информации в таблице остаются неподвижными. Остальные столбцы и строки таблицы не являются фиксированными и при прокрутке могут изменяться.

Свойства VisibleColCount и VisibleRowCount типа Integer содержат *число полностью видимых нефиксированных* столбцов и строк соответственно.

Замечание

Столбцы с частично видимой левой частью и строки с частично видимой верхней частью не входят в число полностью видимых столбцов и строк.

Для указания или анализа ячейки, начиная с которой отображается таблица, можно использовать свойства LeftCol и TopRow типа Longint. Значения этих свойств содержат номера первых видимых столбца и строки соответственно. Управляя значениями свойств LeftCol и TopRow, можно выполнить прокрутку ячеек таблицы.

Например, в процедуре

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  StringGrid1.LeftCol := 3;
  StringGrid1.TopRow := 2;
end;
```

таблица StringGrid1 отображается, начиная с четвертого столбца и третьей строки. Напомним, что нумерация элементов таблицы идет от нуля, однако если сетка содержит по одному фиксированному столбцу и строке, то нумерация остальных, нефиксированных, элементов производится, начиная с единицы.

Свойства Color и FixedColor типа TColor задают, соответственно, *цвета таблицы* и ее *фиксированных элементов*. По умолчанию свойство Color имеет значение clWindow (цвет фона Windows), а свойство FixedColor — значение clBtnFace (цвет кнопки).

Отдельные ячейки таблицы могут разделяться линиями сетки. Толщину линий сетки в пикселах задает свойство GridLineWidth типа Integer, по умолчанию его значение равно 1.

Для доступа к отдельной ячейке служит свойство Cells[ACol,ARow: Integer] типа String, являющееся двумерным массивом, каждый элемент которого есть строка. Индекс ACol определяет столбец, а индекс ARow — строку ячейки в таблице. Напомним еще раз, что нумерация столбцов и строк начинается с нуля. Попытка обращения к несуществующей ячейке не приводит к ошибке, но операция не выполняется. Свойство Cells можно использовать только во время выполнения программы, оно доступно для чтения и записи.

Так, после выполнения команд

```
Label1.Caption := StringGrid1.Cells[5,9];
StringGrid1.Cells[1,2] := 'Hello';
```

надпись Label1 будет отображать содержимое ячейки, находящейся на пересечении шестого столбца и десятой строки таблицы StringGrid1, а во вторую ячейку третьей строки таблицы запишется строка Hello.

Свойства Cols[Index: Integer] и Rows[Index: Integer] типа TStrings обеспечивают *доступ* к любому столбцу и строке таблицы соответственно. Значение свойства Cols[Index] представляет собой список строк, содержащих значения всех ячеек столбца с номером Index, а значение свойства Rows[Index] — список строк, содержащих значения всех ячеек строки с номером Index. Содержимое ячеек *фиксированных* столбцов и строк также входит в значения свойств Cols и Rows.

Например, в результате выполнения процедуры

```
ListBox1.Items.Assign(StringGrid1.Cols[3]);
```

копируется информация из одного списка (ячеек четвертого столбца таблицы StringGrid1) в другой (ListBox1) с заменой содержимого последнего. Если размеры списков не совпадают, то после замены число элементов заменяемого списка становится равным числу элементов копируемого списка.

Компонент StringGrid способен автоматически отображать в своих ячейках текстовую информацию. При необходимости вывести в отдельных ячейках, столбцах или строках нетекстовые данные или текстовые данные совместно с графикой программист должен использовать обработчик события OnDrawCell типа TdrawCellEvent, которое возникает при прорисовке любой ячейки. Тип события OnDrawCell описан так:

```
type TDrawCellEvent = procedure(Sender: TObject; ACol, ARow: Longint;
      Rect: TRect; State: TGridDrawState) of object;
```

Параметры ACol и ARow определяют соответственно *столбец* и *строку*, в которых находится прорисовываемая ячейка. Параметр Rect содержит координаты ограничивающе-

го ячейку прямоугольника. Параметр `State` определяет состояние ячейки и может принимать комбинации следующих значений:

- `gdSelected` (ячейка находится в выбранном диапазоне);
- `gdFocused` (ячейка имеет фокус ввода);
- `gdFixed` (ячейка находится в фиксированном диапазоне).

Порядок вызова события `OnDrawCell` зависит от значения свойства `DefaultDrawing` типа `Boolean`. Если свойство имеет значение `True` (по умолчанию), то перед генерацией события `OnDrawCell` в ячейке отображается фон и выводится текстовая информация. Затем вокруг выбранной ячейки рисуется прямоугольник выбора. Если свойство `DefaultDrawing` имеет значение `False`, то сразу вызывается событие `OnDrawCell`, в обработчике которого следует предусмотреть операции по прорисовке области таблицы `StringGrid`.

Замечание

Событие `OnDrawCell` генерируется всегда, независимо от значения свойства `DefaultDrawing`. Соответственно, инструкции, расположенные в обработчике этого события, также вызываются всегда, когда производится прорисовка ячеек таблицы. Поэтому удобно устанавливать свойство `DefaultDrawing` в значение `True`, что обеспечит прорисовку таблицы по умолчанию, а в обработчике события `OnDrawCell` для требуемых ячеек самостоятельно отображать необходимую информацию. Так можно поступить, когда требуется, например, разместить в некоторых ячейках рисунки или выделить цветом ту или иную ячейку.

Эта особенность отображения отличает компонент `StringGrid` от некоторых других, например от компонента-списка `ListBox`. У списка `ListBox` событие `OnDrawItem`, в обработчике которого программно можно выполнить собственный вывод строк списка на экран, генерируется в случае, если способ (свойство `Style`) прорисовки компонента отличается от стандартного. Поэтому при необходимости изменить отображение отдельных элементов списка нужно программно прорисовывать все строки списка. Для таблицы `StringGrid` такого не происходит, и программисту нужно самостоятельно прорисовывать только те ячейки, которые действительно требуют этого.

Далее приводится пример процедуры программной прорисовки ячеек таблицы:

```
// Значение свойства DefaultDrawing должно быть установлено в True
procedure TForm1.StringGrid1DrawCell(Sender: TObject; ACol,
   ARow: Integer; Rect: TRect; State: TGridDrawState);
begin
  if (ACol = 3) and (ARow = 2) then begin
    StringGrid1.Canvas.Brush.Color := clRed;
    StringGrid1.Canvas.FillRect(Rect);
    ImageList1.Draw(StringGrid1.Canvas, Rect.Left + 2, Rect.Top + 2, 7);
    StringGrid1.Canvas.Font.Color := clYellow;
    StringGrid1.Canvas.Font.Style := [fsItalic];
    StringGrid1.Canvas.TextOut(Rect.Left + ImageList1.Width + 4,
        Rect.Top + 2, StringGrid1.Cells[3, 2]);
  end;
  if gdFocused in State then begin
    StringGrid1.Canvas.Brush.Color := clBlue;
```

```
    StringGrid1.Canvas.FillRect(Rect);
    StringGrid1.Canvas.Font.Color := clBlack;
    StringGrid1.Canvas.Font.Size := 14;
    StringGrid1.Canvas.Font.Style := [fsBold];
    StringGrid1.Canvas.TextOut(Rect.Left, Rect.Top,
       StringGrid1.Cells[StringGrid1.Col, StringGrid1.Row]);
  end;
end;
```

Фон ячейки, лежащей на пересечении третьего столбца и четвертой строки, окрашен в красный цвет. Текст в этой ячейке выводится желтым цветом и курсивом. Слева от текста выводится восьмой из находящихся в компоненте `ImageList1` рисунков. Список с рисунками можно подготовить заранее при разработке приложения и загрузить динамически в процессе выполнения приложения. Ячейка, имеющая фокус ввода, окрашена в синий цвет, а текст в ней выводится шрифтом с полужирным начертанием и размером в 14 пунктов. Отображение остальных ячеек выполняется обычным способом.

При прорисовке ячеек используются свойство `Canvas` элемента `StringGrid1` и параметр `Rect`. Если обработчик события `OnDrawCell` является общим для нескольких таблиц `StringGrid`, то при прорисовке их ячеек вместо названия конкретного компонента (в примере это `StringGrid1`) необходимо ставить конструкцию `(Sender as TStringGrid)` или `TStringGrid(Sender)`. Например, инструкция

```
(Sender as TStringGrid).Canvas.Font.Color := clRed;
```

некоторой процедуры обработки устанавливает красный цвет символов ячеек таблицы, указываемой параметром `Sender`.

Аналогичным способом можно выделять не только заданную ячейку, но и целые столбцы или строки.

Доступ к параметрам таблицы для их настройки возможен через свойство `Options` типа `TGridOptions`. Это свойство представляет собой множество и может принимать комбинации следующих значений:

- `goFixedVertLine` и `goFixedHorzLine` — отображение в сетке для фиксированных элементов вертикальных и горизонтальных разделительных линий;
- `goVertLine` и `goHorzLine` — отображение в сетке вертикальных и горизонтальных разделительных линий;
- `goRangeSelect` — пользователю разрешен выбор диапазона ячеек; игнорируется, если установлено значение `goEditing`;
- `goDrawFocusSelected` — содержащая фокус ввода ячейка выделяется прямоугольной рамкой и цветом;
- `goRowSizing` и `goColSizing` — допускается изменять высоту прокручиваемых строк и ширину прокручиваемых столбцов соответственно;
- `goRowMoving` и `goColMoving` — допускается перемещать с помощью мыши прокручиваемые строки и столбцы;
- `goEditing` — пользователю разрешается редактировать данные в ячейках;
- `goTabs` — допускается перемещение между ячейками с помощью клавиш <Tab> и <Shift>+<Tab>;

- goRowSelect — выбирается вся строка; если задано это значение, то установка goAlwaysShowEditor не действует;
- goAlwaysShowEditor — сетка не блокирует режим редактирования, и пользователь не должен нажимать клавишу <F2> (действие этой клавиши зависит от значения свойства EditorMode) или выполнять двойной щелчок мышью на ячейке для перехода в режим редактирования данных в ячейке; если не установлено значение goEditing или установлено значение goRowSelect, то данная установка не действует;
- goThumbTracking — данные в ячейках обновляются в процессе прокручивания таблицы; если это значение не задано, то обновление данных при прокрутке происходит после отпускания ползунка полосы прокрутки.

По умолчанию для сетки строк установлены значения goFixedVertLine, goFixedHorzLine, goVertLine, goHorzLine и goRangeSelect.

Пользователь может вводить или редактировать данные в ячейках, когда установлено значение goEditing. При этом переход в режим редактирования можно выполнить следующими способами:

- автоматически, если установлено значение goAlwaysShowEditor;
- двойным щелчком мыши на ячейке;
- нажатием клавиши <F2>, если свойство EditorMode установлено в значение True.

При *редактировании* содержимого ячеек возникают события OnGetEditMask типа TGetEditEvent и OnSetEditText типа TSetEditEventa: первое возникает при попытке редактирования, а второе — после завершения изменения ячеек. Программист может использовать эти события для управления данными, которые находятся в таблице и редактируются пользователем.

Замечание

Не допускается визуально редактировать данные ячеек, принадлежащих фиксированным столбцам или строкам. Однако доступ на программном уровне разрешен ко всем ячейкам таблицы, в том числе и к фиксированным.

При *выборе* некоторой ячейки таблицы свойства Row и Col типа Longint содержат номер строки и столбца этой ячейки соответственно. Свойства доступны для записи, что можно использовать для программного выбора ячейки при выполнении приложения.

Перед *выбором* ячейки возникает событие OnSelectCell типа TSelectCellEvent, описываемого следующим образом:

```
type TSelectCellEvent = procedure (Sender: TObject; ACol, ARow: Longint;
    var CanSelect: Boolean) of object;
```

Параметры ARow и ACol указывают выбираемую ячейку, а параметр CanSelect определяет возможность перехода к этой ячейке. Если переход к ячейке нужно заблокировать, то параметр CanSelect устанавливается в значение False. Если его значение True (по умолчанию), то выполняется выбор указанной ячейки.

В качестве примера рассмотрим программу, в которой выполняется пересчет денежной суммы. Исходная сумма задается в рублях и переводится затем в различные валюты. На рис. 8.11 показано окно программы при ее выполнении.

Рис. 8.11. Окно программы пересчета денежной суммы в рублях в другие валюты

В листинге 8.4 приводится текст модуля `uExt`, реализующего форму `Form1` приложения.

Листинг 8.4. Программа пересчета денежных сумм

```
unit uExt;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Grids;
type  TForm1 = class(TForm)
    btnCalculation: TButton;
           lblDate: TLabel;
           btnExit: TButton;
        sgCurrency: TStringGrid;
         edtAmount: TEdit;
         lblAmount: TLabel;
           lblRate: TLabel;
           edtRate: TEdit;
       lblCurrency: TLabel;
            btnAdd: TButton;
         btnDelete: TButton;

    procedure btnExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnCalculationClick(Sender: TObject);
    procedure edtAmountKeyPress(Sender: TObject; var Key: Char);
    procedure edtRateKeyPress(Sender: TObject; var Key: Char);
    procedure btnAddClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
implementation
{$R *.DFM}

// Большинство действий, заданных в процедуре,
// можно выполнить через Инспектор объектов
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Установка свойств таблицы
  sgCurrency.ColCount := 3;
  sgCurrency.RowCount := 5;
  sgCurrency.FixedCols := 0;
  sgCurrency.FixedRows := 1;
  sgCurrency.Cells[0,0] := 'Валюта';
  sgCurrency.Cells[1,0] := 'Курс к доллару';
  sgCurrency.Cells[2,0] := 'Сумма';
  sgCurrency.ColWidths[0] := 160;
  sgCurrency.ColWidths[1] := 90;
  sgCurrency.ColWidths[2] := 120;
  sgCurrency.Cells[0,1] := 'Американский доллар';
  sgCurrency.Cells[0,2] := 'Евро';
  sgCurrency.Cells[0,3] := 'Финская марка';
  sgCurrency.Cells[0,4] := 'Английский фунт стерлингов';
  sgCurrency.Cells[1,1] := '1';
  sgCurrency.Cells[1,2] := '2' + DecimalSeparator + '18';
  sgCurrency.Cells[1,3] := '6' + DecimalSeparator + '63';
  sgCurrency.Cells[1,4] := '0' + DecimalSeparator + '69';
  sgCurrency.Options := [goFixedVertLine, goFixedHorzLine, goVertLine,
     goHorzLine, goRangeSelect, goAlwaysShowEditor, goEditing];
  // Установка свойств других элементов интерфейса
  lblDate.Caption := 'По состоянию на ' + DateToStr(Date);
  lblAmount.Caption := '&Сумма в рублях';
  lblAmount.FocusControl := edtAmount;
  edtAmount.Text := '';
  lblRate.Caption := '&Курс рубля к доллару';
  lblRate.FocusControl := edtRate;
  edtRate.Text := '28' + DecimalSeparator + '86';
  lblCurrency.Caption := 'Сумма в &Валюте';
  lblCurrency.FocusControl := sgCurrency;
  btnCalculation.Default := True;
end;

procedure TForm1.btnCalculationClick(Sender: TObject);
var                                   n: longint;
    Amount, Currency, Rate, CurrencyRate: real;
begin
  // Преобразование суммы и курса в вещественные числа.
  // В случае ошибки — выдача сообщения и выход из процедуры
  try
    Amount := StrToFloat(edtAmount.Text);
    Rate := StrToFloat(edtRate.Text);
```

```
  except
    MessageDlg('Ошибка ввода начальных данных!', mtError, [mbOK], 0);
    edtAmount.SetFocus;
    Raise EAbort.Create('');
  end;
  // Расчет значения суммы для каждой строки таблицы
  // и запись этого значения в соответствующую ячейку.
  // В процессе вычислений обрабатываются возможные ошибки
  for n := 1 to sgCurrency.RowCount — 1 do begin
    try
       CurrencyRate := StrToFloat(sgCurrency.Cells[1,n]);
       Currency := Amount / Rate * CurrencyRate;
       sgCurrency.Cells[2,n] := FloatToStrF(Currency, ffFixed, 10, 2);
    except
       MessageDlg('Ошибка расчета!', mtError, [mbOK], 0);
       sgCurrency.Cells[2,n] := '?';
    end;
  end;
end;

procedure TForm1.edtAmountKeyPress(Sender: TObject; var Key: Char);
begin
  // Вводить только символы, допустимые для вещественного числа,
  // и обрабатывать нажатия клавиши <BackSpace>
  if not (Key in [#8,'0'..'9','+','E','e',DecimalSeparator]) then Key := #0;
end;

procedure TForm1.edtRateKeyPress(Sender: TObject; var Key: Char);
begin
  // Вводить только символы, допустимые для вещественного числа,
  // и обрабатывать нажатия клавиши <BackSpace>
  if not (Key in [#8,'0'..'9','+','E','e',DecimalSeparator]) then Key := #0;
end;

procedure TForm1.btnAddClick(Sender: TObject);
begin
  // Добавление новой строки в конец таблицы
  sgCurrency.RowCount := sgCurrency.RowCount + 1;
  // Очистка ячеек добавленной строки
  sgCurrency.Rows[sgCurrency.RowCount — 1].Clear;
end;

procedure TForm1.btnDeleteClick(Sender: TObject);
var n :longint;
begin
  // Если осталось две строки, операцию удаления не выполнять
  if sgCurrency.RowCount = 2 then exit;
  // Сдвиг строк вверх, начиная со строки, содержащей выделенную
  for n := sgCurrency.Row to sgCurrency.RowCount — 2 do
    sgCurrency.Rows[n] := sgCurrency.Rows[n + 1];
```

```
  // Удаление последней строки таблицы
  sgCurrency.RowCount := sgCurrency.RowCount - 1;
end;

procedure TForm1.btnExitClick(Sender: TObject);
begin
  Close;
end;
end.
```

Центральное место в форме занимает таблица строк `sgCurrency`, имеющая три столбца и пять строк (первоначально), верхняя из которых фиксированная и содержит наименования столбцов. Первый столбец заполняется названиями валют, во втором указывается курс (коэффициент) валюты по отношению к доллару США, а в третий столбец заносятся суммы, эквивалентные денежной сумме в рублях, задаваемой в специальном поле ввода.

В правой части окна расположены два однострочных редактора, представляющие собой поля для ввода суммы в рублях (редактор `edtAmount`) и курса рубля по отношению к доллару (редактор `edtRate`). При вводе осуществляется предварительный контроль допустимости вводимых с клавиатуры символов. С этой целью для обоих редакторов подключены обработчики события `OnKeyPress`, которые не позволяют пользователю ввести недопустимый для вещественного числа символ. Такая проверка не гарантирует от всех возможных ошибок набора, т. к. в редактор можно ввести, например, код `++123`, не являющийся допустимым для указания суммы и курса числом.

При нажатии кнопки **Расчет** (кнопка `btnCalculation`) на основе данных из двух полей ввода и первых двух столбцов таблицы осуществляется перерасчет соответствующей каждому виду валюты суммы, которая записывается в ячейку третьего столбца. При этом выполняется контроль возможных ошибок.

Пользователю разрешено редактировать данные в ячейках, для чего свойству `Options` таблицы задано значение `goEditing`. При необходимости в конец таблицы можно добавить новую строку нажатием кнопки **Добавить** (кнопка `btnAdd`). Кнопка **Удалить** (кнопка `btnDelete`) удаляет строку, в которой находится выбранная ячейка. При этом расположенные ниже строки сдвигаются вверх, а освободившаяся последняя строка удаляется. В случае, если остались только две строки, удаление не выполняется.

В нескольких местах программы используется глобальная переменная `DecimalSeparator` из модуля `SysUtils`. Эта переменная имеет тип `char` и содержит символ-разделитель целой и дробной частей числа. Символ-разделитель зависит от установки в окне **Язык и стандарты** Панели управления Windows. Если программа не использует переменную `DecimalSeparator`, а записывает какой-либо конкретный разделитель, то на другом компьютере может произойти ошибка при попытке преобразования строки в число. Исключение `EConvertError` будет генерироваться, например, когда в редактор заносится строка `100.00`, а в качестве разделителя установлена запятая (`,`). В этом случае для обеспечения работоспособности программы пользователь может изменить настройку системы через Панель управления Windows. Однако более пра-

вильным подходом будет, если разработчик, предвидя такую ситуацию, использует в приложении глобальную переменную `DecimalSeparator`.

В верхней части формы выводится информационная строка с текущей датой.

Элементы с вкладками

Элемент с вкладками представляет собой элемент управления, содержащий несколько вкладок (страниц, листов), на которые пользователь может переходить, щелкая мышью на *ярлыках* этих страниц. Для работы с содержащими вкладки элементами в Delphi служат компоненты `TabControl` и `PageControl`, размещенные на странице **Win32** Палитры компонентов. Кроме того, на странице **Win 3.1** Палитры компонентов находятся три компонента от предыдущих версий — `TabSet`, `TabbedNotebook` и `Notebook`, используемые для обеспечения совместимости с программами, разработанными в среде Windows 3.*x*.

Компоненты `TabControl` и `PageControl` являются контейнерами и могут содержать в себе другие компоненты, объединяя и группируя их. Оба компонента происходят от общего класса `TCustomTabControl` и во многом похожи, хотя имеют и существенные различия в поведении и использовании.

Одностраничный блокнот

Одностраничный блокнот `TabControl`, расположенный на странице **Win32** Палитры компонентов, представляет собой прямоугольную область с набором вкладок. При выборе пользователем какой-либо вкладки происходит автоматическое переключение на нее. Одностраничный блокнот обычно применяется в случаях, когда требуется набор вкладок, по функциям совпадающий с группой зависимых переключателей, например, при создании календаря или записной книжки с алфавитным указателем.

Компонент `TabControl` при отображении может иметь различные *стили* (рис. 8.12), определяемые свойством `Style` типа `TTabStyle`, принимающим следующие значения:

- `tsTabs` (стандартные вкладки объемного вида) — по умолчанию;
- `tsButtons` (вкладки в виде кнопок);
- `tsFlatButtons` (вкладки в виде плоских кнопок).

Рис. 8.12. Компоненты `TabControl` с различными стилями

Если вкладки имеют вид кнопок, то у компонента `TabControl` отсутствует рамка, ограничивающая страницу. При необходимости программист должен сам *визуально огра-*

ничить область указанного компонента. Для этого можно использовать такие компоненты, как `Bevel` или `Panel`, а также средства класса `TCanvas`.

Число и *названия* вкладок определяет свойство `Tabs` типа `TStrings`, представляющее собой список строк, используемый для формирования вкладок. При добавлении к списку `Tabs` новой строки автоматически создается новая вкладка с этим именем. Так как свойство `Tabs` доступно через Инспектор объектов, то при конструировании приложения вкладки создаются и настраиваются с помощью Редактора строк (String List editor). При выполнении программы работать с вкладками можно так же, как с любым объектом типа `TStrings`, настраивая свойства и вызывая соответствующие методы, например, `Add` для добавления новой вкладки, `Delete` для удаления существующей и т. д.

Так, в примере

```
TabControl1.Tabs[2] := 'Новое название';
TabControl1.Tabs.Add('Новая вкладка');
TabControl1.Tabs.Delete(7);
```

к компоненту `TabControl1` добавляется новая вкладка, удаляется восьмая вкладка и изменяется заголовок третьей вкладки.

Свойство `MultiLine` типа `Boolean` определяет, могут ли ярлыки вкладок отображаться в *несколько строк*. Если свойство имеет значение `False` (по умолчанию), то ярлыки выводятся одной строкой, и если они не умещаются в одной строке, то в правой части элемента управления появляются стрелки, с помощью которых можно выполнить прокрутку ярлыков. На рис. 8.12 ярлыки, расположенные в одну строку, имеют средний и правый компоненты `TabControl`. Если установить свойство `MultiLine` в значение `True`, то ярлыки вкладок можно отобразить в несколько строк (левый компонент `TabControl` на рис. 8.12).

Расположение вкладок в одностраничном блокноте определяется свойством `TabPosition` типа `TTabPosition`, которое может принимать следующие значения:

- `tpTop` (вверху) — по умолчанию;
- `tpBottom` (внизу);
- `tpLeft` (слева);
- `tpRight` (справа).

При размещении ярлыков вкладок в несколько строк, когда свойство `MultiLine` имеет значение `True`, свойство `ScrollOpposite` типа `Boolean` определяет *поведение вкладок при выборе*. При установке свойства `ScrollOpposite` в значение `True` строка с выбранной вкладкой перемещается в первый ряд. По умолчанию свойство имеет значение `False`, и перемещение вкладок не происходит. Если свойство `ScrollOpposite` устанавливается в значение `True`, то одновременно в значение `True` устанавливается и свойство `MultiLine`.

Размеры вкладок задаются свойствами `TabWidth` и `TabHeight` типа `Smallint`, определяющими их ширину и высоту в пикселах. По умолчанию оба свойства имеют значение `0`, что соответствует автоматическому изменению размеров вкладки в зависимости от заголовка.

Свойство `TabIndex` типа `Integer` *указывает выбранную вкладку* в массиве строк. Это свойство доступно для записи и может быть использовано программистом для пере-

ключения на вкладку с нужным номером. Например, для переключения на вторую вкладку можно выполнить следующую инструкцию:

```
TabControl1.TabIndex := 1;
```

Напомним, что нумерация строк в списке начинается с нуля. Если не выбрана ни одна вкладка (по умолчанию), то свойство `TabIndex` имеет значение `-1`. После динамического удаления вкладки (при выполнении программы) ни одна из вкладок не будет выбрана. Переключение вкладок при разработке приложения выполняется с помощью Инспектора объектов путем изменения значения свойства `TabIndex`.

Свойство `HotTrack` типа `Boolean` определяет, *подсвечивается ли заголовок* вкладки, когда над ним находится указатель мыши. По умолчанию свойство `HotTrack` имеет значение `False`, и заголовки вкладок при перемещении по ним указателя мыши цветом не выделяются.

Вкладки, кроме текста, могут отображать рисунок. Операции, связанные с выводом рисунков, практически не отличаются от аналогичных операций для визуальных компонентов, например, для кнопок `ToolButton` панели инструментов `ToolBar`. Программист помещает в форму компонент `ImageList`, заполняет его рисунками, после чего указывает на сформированный список рисунков значением свойства `Images` типа `TCustomImageList`. Вкладки получают рисунки из указанного списка соответственно их номерам в списке строк `Tabs`.

Так как все вкладки имеют одну страницу (общую область отображения), то при *переключении вкладок* программист должен предусмотреть действия, связанные с обновлением информации в этой области. Для такой обработки удобно использовать события `OnChange` типа `TNotifyEvent`, генерируемое при *активизации* вкладки, и `OnChanging` типа `TTabChangingEvent`, возникающее непосредственно *перед* активизацией.

Рассмотрим пример использования одностраничного блокнота для создания простого календаря. Для выбора номера месяца и отображения числа дней в нем использован элемент `tcMonthDays` класса `TTabControl`. Он содержит двенадцать ярлыков (по числу месяцев в году), и в его области расположены две надписи `lblMonthDays` и `lblDate` (рис. 8.13).

Рис. 8.13. Вывод количества дней в месяце

В листинге 8.5 приведен текст модуля `uExt2`, который реализует главную форму приложения.

Листинг 8.5. Пример использования одностраничного блокнота

```
unit uExt2;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, Grids, DBGrids, Db, DBTables, ComCtrls, StdCtrls;
type
  TForm2 = class(TForm)
     tcMonthDays: TTabControl;
    lblMonthDays: TLabel;
         lblDate: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure tcMonthDaysChange(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var      Form2: TForm2;
    yy, mm, dd: word;
implementation
{$R *.DFM}

procedure TForm2.FormCreate(Sender: TObject);
var n: integer;
begin
  // Вывод текущей даты
  lblDate.Caption := 'Дата: ' + DateToStr(Date);
  // Формирование вкладок и расположение их ярлыков в несколько строк
  tcMonthDays.Multiline := True;
  tcMonthDays.Tabs.Clear;
  for n := 1 to 12 do
    tcMonthDays.Tabs.Add(IntToStr(n));
  // Определение текущего месяца
  DecodeDate(Date, yy, mm, dd);
  // Выбор вкладки, соответствующей текущему месяцу
  tcMonthDays.TabIndex := mm - 1;
  // Обновление информации в области вывода
  tcMonthDaysChange(Sender);
end;

procedure TForm2.tcMonthDaysChange(Sender: TObject);
begin
  // Обновление информации в области вывода
  case tcMonthDays.TabIndex + 1 of
    1,3,5,7,8,10,12: lblMonthDays.Caption := 'В этом месяце 31 день.';
    4,6,9,11:        lblMonthDays.Caption := 'В этом месяце 30 дней.';
    2:               if (yy mod 4 = 0) and (not(yy mod 100 = 0) or
                                            (yy mod 1000 = 0))
```

```
                then lblMonthDays.Caption := 'В этом месяце 29 дней.'
                else lblMonthDays.Caption := 'В этом месяце 28 дней.';
  end;
end;
end.
```

После запуска приложения в процедуре `DecodeDate` из модуля `SysUtils` определяется номер текущего месяца. После этого выбирается соответствующая вкладка и в надписи `lblMonthDays` выводится количество дней. Надпись `lblDate` показывает текущую дату.

При выборе другой вкладки в обработчике события `OnChange` выполняется обновление информации в области вывода компонента `tcMonthDays`.

Событие `OnChanging` происходит до перехода (переключения) на другую вкладку, поэтому его обработчик позволяет *запретить переключение*, если не выполнены какие-либо необходимые условия. Тип `TTabChangingEvent` этого события описан следующим образом:

```
type TTabChangingEvent = procedure(Sender: TObject;
                                   var AllowChange: Boolean) of object;
```

Параметр `AllowChange` определяет возможность переключения вкладки. Если `AllowChange` имеет значение `True`, то с вкладки можно перейти на другую, в противном случае — нет.

Пример блокировки переключения вкладок:

```
procedure TForm2.TabControl2Changing(Sender: TObject;
                                     var AllowChange: Boolean);
begin
  if length(Edit1.Text) = 0
      then AllowChange := False
      else AllowChange := True;
  // Присвоить значение параметру AllowChange
  // можно более коротким способом:
  // AllowChange := length(Edit1.Text) <> 0;
end;
```

В области элемента управления `TabControl2` находится редактор `Edit1`. Если поле этого редактора содержит пустое значение, то переход на другую вкладку блокируется. Компонент `Edit1` может быть расположен и вне области компонента `TabControl2`.

Для управления отображением данных в компоненте `TabControl` иногда требуется знать размер его страницы. Страница является клиентской областью блокнота, ее размеры определяет свойство `DisplayRect` типа `TRect`, позволяющее получить (в пикселах) координату левого верхнего угла, ширину и высоту области.

В клиентской области элемента `TabControl` можно размещать не только элементы управления, но и различные графические изображения. Методы рисования подробно рассматриваются в *главе 10*, посвященной графическим возможностям Delphi. Здесь мы скажем только, что для рисования на поверхности визуальных компонентов используются средства класса `TCanvas` (свойство `Canvas`).

Замечание

Компонент `TabControl` не имеет доступного свойства `Canvas`. Поэтому для рисования на его поверхности можно поместить в его клиентскую область другой компонент, обладающий таким свойством, как, например, `Image` или `PaintBox`, и далее выводить графику уже на поверхности этого компонента.

Более тонким способом вывода графики является применение свойства `Handle` элемента `TabControl`. Это свойство оконных элементов управления обеспечивает доступ к дескриптору окна, что можно использовать для вызова функций API, в том числе графических.

В следующем примере реализуется одностраничный блокнот с картинками (листинг 8.6).

Листинг 8.6. Пример одностраничного блокнота с картинками

```
unit uTabCon;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ComCtrls, StdCtrls;
type
  TForm5 = class(TForm)
    tcGraphics: TTabControl;
    procedure tcGraphicsChange(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form5: TForm5;
implementation
{$R *.DFM}

procedure TForm5.tcGraphicsChange(Sender: TObject);
var CanvasTC: TCanvas;
begin
  try
    // Создание объекта CanvasTC типа TCanvas
    CanvasTC := TCanvas.Create;
    try
      // Получение ссылки на область вывода компонента tcGraphics
      CanvasTC.Handle := GetDC(tcGraphics.Handle);
      // Вывод геометрических фигур
      case tcGraphics.TabIndex of
        0: begin
             CanvasTC.Brush.Color := clGreen;
             CanvasTC.FillRect(tcGraphics.DisplayRect);
             CanvasTC.Pen.Color := clRed;
             CanvasTC.Ellipse(30, 50, 130, 150);
           end;
```

```
        1: begin
             CanvasTC.Brush.Color := clWhite;
             CanvasTC.FillRect(tcGraphics.DisplayRect);
             CanvasTC.Pen.Color := clBlue;
             CanvasTC.Rectangle(40, 70, 120, 150);
           end;
      end;
    finally
      // Уничтожение ссылки на область вывода компонента tcGraphics
      ReleaseDC(tcGraphics.Handle, CanvasTC.Handle);
    end;
  finally
    // Уничтожение объекта CanvasTC
    CanvasTC.Free;
  end;
end;
end.
```

При выборе вкладок одностраничного блокнота `tcGraphics` в его области рисуются геометрические фигуры. При выборе вкладки **Круг** выводится красная окружность на зеленом фоне, при выборе вкладки **Квадрат** отображается синий квадрат на белом фоне. Первоначально после запуска приложения выбирается первая вкладка, но ни одна фигура не выводится.

Еще один пример использования компонента `TabControl` будет приведен при рассмотрении приложений для работы с базами данных.

Многостраничный блокнот

Многостраничный блокнот `PageControl`, также расположенный на странице **Win32** Палитры компонентов, в отличие от компонента `TabControl`, состоит из нескольких *страниц (вкладок)*, расположенных одна под другой. Каждая страница имеет ярлык и относительно независима от других страниц. Компактное расположение страниц блокнота позволяет удобно размещать и группировать другие элементы управления. При выборе ярлыка автоматически выбирается и соответствующая страница, после чего для пользователя становятся доступными расположенные на ней элементы управления. В отличие от компонента `TabControl`, для многостраничного блокнота ярлык является частью страницы. Часто компонент `PageControl` используется для создания окон свойств. На рис. 8.14 в качестве примера приводится окно свойств табличного процессора Microsoft Excel.

По своему внешнему виду компонент `PageControl` не отличается от компонента `TabControl` и имеет с ним некоторые общие элементы, например, свойства `Style`, `Images`, `Multiline`, `ScrollOpposite`, `HotTrack`, `TabWidth` и `TabHeight`, события `OnChange` и `OnChanging`. Но многостраничный блокнот является более сложным элементом управления, чем компонент `TabControl`. Отдельные страницы многостраничного блокнота называют также *панелями*, а сам компонент `PageControl` — *множественной панелью*.

Число страниц многостраничного блокнота указывает свойство `PageCount` типа `Integer`, действующее во время выполнения программы и доступное только для чтения.

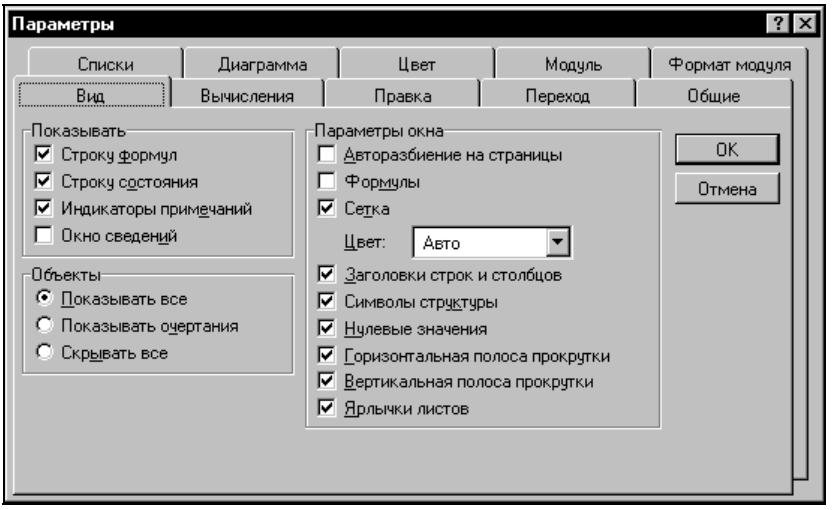


Рис. 8.14. Окно свойств табличного процессора Microsoft Excel

Свойство `ActivePage` типа `TTabSheet` определяет *выбранную вкладку*, а вместе с ней и страницу, находящуюся на переднем плане. Это свойство доступно для записи и для чтения. При разработке приложения в качестве значения свойства `ActivePage` Инспектор объектов отображает только заголовок активной страницы.

При создании приложения программист может переходить на другие страницы, щелкая на их ярлыках мышью, однако, в отличие от компонента `TabControl`, переключение страниц компонента `PageControl` программным способом является более трудной и не слишком удобной для разработчика задачей. Написанный по аналогии со свойством `TabIndex` элемента `TabControl` код

```
PageControl.ActivePage.PageIndex := 1;
```

не означает переход на вторую страницу блокнота. В результате выполнения этой инструкции активная страница действительно получит новый номер, равный `1` (если он до этого был другим), и по-прежнему останется активной, т. е. произойдет только перенумерация страниц.

Одним из вариантов переключения страниц многостраничного блокнота является описание специальной переменной подходящего типа, создание ее экземпляра и заполнение списком страниц. После этого активную страницу можно выбрать в списке по заданному номеру. При удалении или добавлении страницы содержащийся в переменной список должен формироваться заново. Данная техника реализована в следующем примере (листинг 8.7).

Листинг 8.7. Пример переключения страниц многостраничного блокнота

```
unit uPageC3;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls;
```

```
type
  TForm1 = class(TForm)
    PageControl1: TPageControl;
       TabSheet1: TTabSheet;
       TabSheet2: TTabSheet;
       TabSheet3: TTabSheet;
           Edit1: TEdit;
         Button3: TButton;

    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var  Form1: TForm1;
       TS: TStringList;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var n: integer;
begin
  TS := TStringList.Create;
  for n := 0 to PageControl1.PageCount - 1 do
    TS.AddObject(PageControl1.Pages[n].Name, PageControl1.Pages[n]);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  PageControl1.ActivePage := TTabSheet(TS.Objects[StrToInt(Edit1.Text)]);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  // В данном примере эту инструкцию можно не использовать,
  // при удалении формы список будет удален автоматически
  TS.Free;
end;
end.
```

Для хранения списка страниц используется переменная типа `TStringList`, экземпляр которой создается при создании формы и удаляется вместе с формой.

Замечание

У аналога многостраничного блокнота — компонента `TabbedNotebook` есть более удобный способ программного переключения страниц с помощью свойства `PageIndex`, аналогичного свойству `TabIndex` компонента `TabControl`. Однако компонент `TabbedNotebook` имеет другой недостаток — он не позволяет при разработке приложения переключать страницы щелчком мыши (для этого приходится пользоваться Инспектором объектов).

В многостраничном блокноте `PageControl` при переходе на другую страницу с ярлыком возникают события `OnChange` и `OnChanging`, обработка которых выполняется аналогично обработке одноименных событий компонента `TabControl`. Эти события можно использовать, например, для проверки состояния размещенных на странице элементов или управления доступом к определенным страницам.

Доступ к отдельным страницам многостраничного блокнота осуществляется через свойство `Pages[Index: Integer]` типа `TTabSheet`, представляющее собой список страниц, к которым можно обращаться по номеру в списке. Свойство доступно во время выполнения программы.

Для *активизации* предыдущей или следующей видимой страницы можно использовать метод `SelectNextPage(GoForward: Boolean)`. Параметр `GoForward` определяет направление перехода: если он равен `True`, то активизируется следующая видимая страница, в противном случае — предыдущая.

Похоже работает функция `FindNextPage(CurPage: TTabSheet; GoForward, CheckTabVisible: Boolean): TTabSheet`, которая возвращает *следующую страницу* в многостраничном блокноте. Страницу, с которой начинается поиск, задает параметр `CurPage`. Параметр `CheckTabVisible` определяет диапазон поиска: если он имеет значение `True`, то поиск проводится среди видимых страниц, в противном случае просматриваются все страницы блокнота.

Каждая страница многостраничного блокнота является объектом класса `TTabSheet` и может содержать другие компоненты. Так как страницы являются самостоятельными объектами, для компонента `PageControl` нет специального редактора, как, например, Редактор строк (String List editor), используемый для редактирования вкладок компонента `TabControl`. Поэтому для добавления и удаления страниц используется контекстное меню многостраничного блокнота.

Для *добавления страницы* к многостраничному блокноту при разработке приложения следует щелкнуть на нем правой кнопкой мыши и в появившемся контекстном меню выбрать пункт **New Page** (Новая страница). Контекстное меню также позволяет перейти к следующей или предыдущей странице, но такой переход проще выполнить, щелкнув мышью на нужном ярлыке (Delphi позволяет при разработке приложения переходить на другие страницы с помощью мыши).

Для *удаления страницы* из многостраничного блокнота в контекстном меню нужно выбрать пункт **Delete Page** (Удалить страницу).

Замечание

Перед вызовом контекстного меню должна быть выделена именно удаляемая страница (объект типа `TTabSheet`), а не блокнот (объект типа `TPageControl`), в противном случае произойдет удаление всего компонента `PageControl`.

После перехода на другую страницу выделенным оказывается весь блокнот. Для повторного выделения страницы можно щелкнуть мышью в области страницы (не на ярлыке) или выбрать страницу с помощью Инспектора объектов.

Как и другие объекты, страницы можно создавать и удалять *динамически*. При создании страницы сначала конструктором `Create` создается экземпляр объекта типа `TTabSheet`, затем в свойстве `PageControl` типа `TPageControl` созданной страницы указы-

вается блокнот, которому эта страница будет принадлежать. Удалить страницу позволяет метод Free. Например, в процедуре

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  if PageControl1.PageCount <> 0 then PageControl1.Pages[0].Free;
  with TTabSheet.Create(Self) do begin
     PageControl := PageControl1;
     Caption := 'Новая страница';
  end;
end;
```

при нажатии кнопки Button3 из блокнота PageControl1 динамически удаляется первая страница и добавляется новая. Создание объекта типа TTabSheet выполнено без объявления специальной переменной.

Заголовок ярлыка каждой страницы определяется ее свойством Caption типа TCaption, а рисунок, который может отображаться на ярлыке рядом с заголовком, задается как значение свойства ImageIndex типа Integer. Если рисунок, соответствующий указанному номеру, в компоненте ImageList не найден или значение свойства ImageIndex равно -1, то для этой страницы блокнота рисунок считается отсутствующим. Список рисунков задается свойством Images блокнота. Ширина ярлыка страницы автоматически изменяется по размеру текста и рисунка, отображаемых на ярлыке.

Для красивого и оригинального оформления ярлыков, например, чтобы они отображали текст шрифтами различного цвета и стиля, а также содержали рисунок, можно выполнить *программную прорисовку* области ярлыков. Для этого необходимо установить свойство OwnerDraw типа Boolean блокнота в значение True (по умолчанию False), и вывод содержимого ярлыков всех страниц будет выполняться автоматически. Для кодирования операций прорисовки ярлыков используется событие OnDrawTab типа TDrawTabEvent, возникающее при необходимости перерисовать область ярлыка. Тип TDrawTabEvent описан следующим образом:

```
type TDrawTabEvent = procedure(Control: TCustomTabControl;
    TabIndex: Integer; const Rect: TRect; Active: Boolean) of object;
```

Параметр Control является *ссылкой на объект* типа TCustomTabControl, дочерним типом которого является TPageControl. Использование этого параметра может понадобиться при подготовке обработчика, общего для нескольких компонентов PageControl. Если процедура — обработчик события OnDrawTab предназначена для одного многостраничного блокнота, например, с именем PageControl1, то в теле процедуры вместо параметра Control можно напрямую использовать имя PageControl1.

Параметр TabIndex указывает *номер страницы*, ярлык которой отображается, а параметр Rect содержит *область ярлыка*, в пределах которой можно выполнять операции прорисовки. Параметр Active позволяет определить активность страницы, ярлык которой отображается. Если он имеет значение True, то ярлык принадлежит активной странице и в этом случае можно, например, выделить этот ярлык другим цветом, шрифтом или иным способом.

Рассмотрим программную прорисовку ярлыков на следующем примере. Заголовок ярлыка активной страницы многостраничного блокнота PageControl1 выводится полу-

жирным наклонным шрифтом, слева от текста отображается рисунок в виде стрелки. Заголовки ярлыков неактивных страниц выводятся обычным шрифтом, рисунок имеет вид бледной стрелки (рис. 8.15).

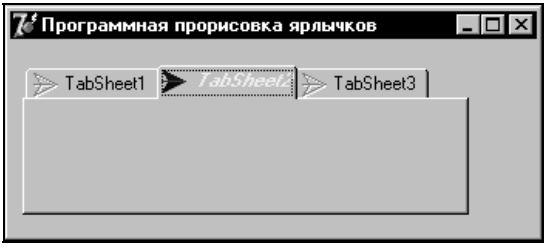


Рис. 8.15. Программная прорисовка ярлыков блокнота

В форме `Form1`, кроме блокнота `PageControl1`, находится компонент `ImageList1`, в который на этапе разработки загружены два рисунка со стрелками: темный рисунок имеет номер `0`, а светлый — `1`. При создании формы свойство `ImageIndex` всех ярлыков блокнота должно указывать на какой-либо из рисунков, содержащихся в списке `ImageList1`. Несмотря на то, что автоматически эти рисунки на ярлыках не отображаются, ссылка на них нужна для того, чтобы ширина ярлыка при программной прорисовке позволила разместить на нем рисунок и текст названия.

Код модуля `uPageC`, реализующего форму `Form1` приложения, приведен в листинге 8.8.

Листинг 8.8. Программная прорисовка ярлыков блокнота

```
unit uPageC;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ImgList, ComCtrls;
type
  TForm1 = class(TForm)
    PageControl1: TPageControl;
       TabSheet1: TTabSheet;
      ImageList1: TImageList;
       TabSheet2: TTabSheet;
       TabSheet3: TTabSheet;

    procedure PageControl1DrawTab(Control: TCustomTabControl;
      TabIndex: Integer; const Rect: TRect; Active: Boolean);
    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  PageControl1.OwnerDraw := True;
  TabSheet1.ImageIndex := 0;
  TabSheet2.ImageIndex := 0;
  TabSheet3.ImageIndex := 0;
end;

// Свойство OwnerDraw компонента PageControl1
// должно быть установлено в значение True
procedure TForm1.PageControl1DrawTab(Control: TCustomTabControl;
  TabIndex: Integer; const Rect: TRect; Active: Boolean);
begin
  if Active then begin
    Control.Canvas.Font.Style := [fsBold, fsItalic];
    Control.Canvas.Font.Color := clYellow;
    Control.Canvas.TextRect(Rect, Rect.Left + 23, Rect.Top + 3,
           (Control as TPageControl).Pages[TabIndex].Caption);
    ImageList1.Draw(Control.Canvas, Rect.Left + 3, Rect.Top + 3, 0);
  end
  else begin
      Control.Canvas.Font.Style := [];
      Control.Canvas.Font.Color := clBlack;
      Control.Canvas.TextRect(Rect, Rect.Left + 23, Rect.Top + 3,
           (Control as TPageControl).Pages[TabIndex].Caption);
      ImageList1.Draw(Control.Canvas, Rect.Left + 3, Rect.Top + 3, 1);
  end;
end;
end.
```

Видимостью отдельных страниц многостраничного блокнота и их содержимого управляют два похожих свойства: `TabVisible` и `Visible` типа `Boolean`. Свойство `TabVisible` определяет, *видимы ли страница и ее ярлык*. По умолчанию оно имеет значение `True`, и страница видима вместе с ярлыком. Программно можно установить свойство `TabVisible` в значение `False`, что приведет к скрытию страницы и ее ярлыка.

В случае, когда страница видима, свойство `Visible` определяет, *видимы ли компоненты*, размещенные на этой странице. По умолчанию это свойство имеет значение `True`, и содержимое страницы видимо. При установке свойства `Visible` в значение `False` содержимое страницы скрывается от пользователя, однако ее ярлык остается попрежнему видимым и позволяет перейти на страницу с помощью мыши. Особенностью свойства `Visible` является то, что содержимое страницы может быть скрыто только после ее активизации. При выборе видимой страницы ее свойство `Visible` снова устанавливается в значение `True`.

Номер страницы в списке страниц блокнота (свойство `Pages`) указывает свойство `PageIndex` типа `Integer`. При изменении номера страницы или удалении какой-либо из страниц номера автоматически пересчитываются. Похожее свойство `TabIndex` типа `Integer` указывает *номер в списке видимых страниц*. Нумерация в списках начинается с нуля.

Каждая страница может иметь свое контекстное меню, которое указывается в свойстве `PopupMenu` типа `TPopupMenu` страницы.

Приведем еще один пример, демонстрирующий работу с компонентом `PageControl`.

При конструировании проекта к многостраничному блокноту (рис. 8.16) добавлены три страницы, на которых размещены компоненты `Button2`, `Edit1` и `Label2`. После запуска приложения все страницы блокнота удаляются вместе с находящимися на них элементами управления и добавляются четыре новые страницы (**Лист 1**, **Лист 2**, **Лист 3** и **Лист 4**). На каждую страницу помещается динамически созданная кнопка `Button`.

Для управления блокнотом в форме расположено семь кнопок, выполняющих операции удаления, добавления и переключения страниц, а также установки свойств видимости. Кнопки **Удалить** и **Добавить** производят операции с активной (выбранной) страницей. Кнопки **Выбрать** и **Видимость** выполняют действия со страницей, номер которой задан в поле с надписью **Номер страницы** (редактор `edtPageNumber`). Номер, введенный в редактор, соответствует нумерации страниц, начиная с единицы, поэтому при использовании этого номера в соответствующих методах его значение уменьшается на единицу.

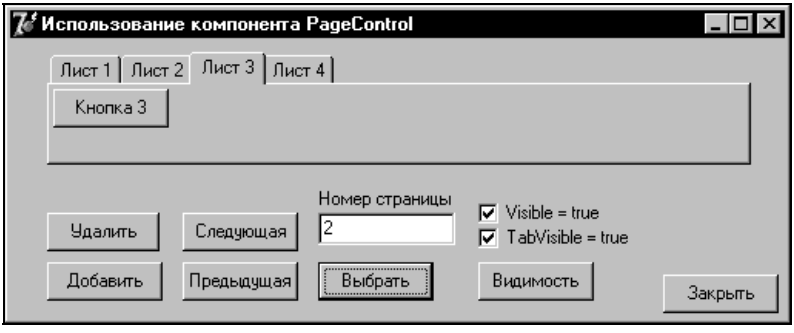


Рис. 8.16. Использование многостраничного блокнота

Описание формы приложения содержится в модуле `Page`, текст которого приведен в листинге 8.9.

Листинг 8.9. Использование компонента `PageControl`

```
unit Page;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls;
```

```
type
  TForm17 = class(TForm)
     PageControl1: TPageControl;
       btnAddPage: TButton;
    btnDeletePage: TButton;
   btnCurrentPage: TButton;
    edtPageNumber: TEdit;
          btnNext: TButton;
         btnPrior: TButton;
       btnVisible: TButton;
           Label1: TLabel;
          btnExit: TButton;
        TabSheet1: TTabSheet;
        cbVisible: TCheckBox;
     cbTabVisible: TCheckBox;
        TabSheet2: TTabSheet;
        TabSheet3: TTabSheet;
          Button2: TButton;
            Edit1: TEdit;
           Label2: TLabel;

    procedure MakeListTab(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnAddPageClick(Sender: TObject);
    procedure btnDeletePageClick(Sender: TObject);
    procedure btnCurrentPageClick(Sender: TObject);
    procedure btnNextClick(Sender: TObject);
    procedure btnPriorClick(Sender: TObject);
    procedure btnVisibleClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var      Form17: TForm17;
      MaxNumber: integer;
             TS: TStringList;
implementation
{$R *.DFM}

procedure TForm17.MakeListTab(Sender: TObject);
var n: integer;
begin
  // Формирование списка страниц блокнота
  TS.Clear;
  for n := 0 to PageControl1.PageCount — 1 do
    TS.AddObject(PageControl1.Pages[n].Name, PageControl1.Pages[n]);
end;
```

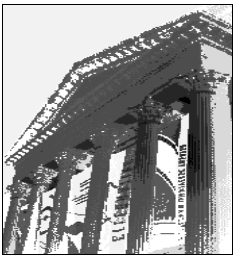
```
procedure TForm17.FormCreate(Sender: TObject);
var   NewPage: TTabSheet;
    NewButton: TButton;
            n: integer;
begin
  cbVisible.Checked := True;
  cbTabVisible.Checked := True;
  MaxNumber := 3;
  PageControl1.Multiline := True;
  // Удаление всех страниц блокнота
  for n := PageControl1.PageCount — 1 downto 0 do
    PageControl1.Pages[n].Destroy;
  // Для добавления страниц использована переменная NewPage типа TTabSheet.
  // Для добавления кнопок использована переменная NewButton типа TButton
  for n := 0 to MaxNumber do begin
    NewPage := TTabSheet.Create(Self);
    NewPage.PageControl := PageControl1;
    NewPage.Caption := 'Лист ' + IntToStr(n + 1);
    NewButton := TButton.Create(Self);
    NewButton.Caption := 'Кнопка ' + IntToStr(n + 1);
    NewButton.Parent := NewPage;
  end;
  // Первоначальное формирование списка страниц блокнота
  TS := TStringList.Create;
  MakeListTab(Sender);
end;

procedure TForm17.btnAddPageClick(Sender: TObject);
begin
  Inc(MaxNumber);
  // Добавление страницы выполнено без объявления специальной переменной
  with TTabSheet.Create(Self) do begin
     PageControl := PageControl1;
     Caption := 'Лист ' + IntToStr(MaxNumber + 1);
  end;
  // Формирование списка страниц блокнота заново
  MakeListTab(Sender);
end;

procedure TForm17.btnDeletePageClick(Sender: TObject);
begin
  if PageControl1.PageCount <= 0 then exit;
  PageControl1.Pages[PageControl1.ActivePage.PageIndex].Free;
  // Формирование списка страниц блокнота заново
  MakeListTab(Sender);
end;

procedure TForm17.btnCurrentPageClick(Sender: TObject);
begin
  // Номера страниц в редакторе edtPageNumber отсчитываются от единицы
  PageControl1.ActivePage:=TTabSheet(TS.Objects[StrToInt(edtPageNumber.Text)-1]);
end;
```

```
procedure TForm17.btnNextClick(Sender: TObject);
begin
  PageControl1.SelectNextPage(True);
end;

procedure TForm17.btnPriorClick(Sender: TObject);
begin
  PageControl1.SelectNextPage(False);
end;

procedure TForm17.btnVisibleClick(Sender: TObject);
begin
  if cbVisible.Checked
    then PageControl1.Pages[StrToInt(edtPageNumber.Text)-1].TabVisible:=True
    else PageControl1.Pages[StrToInt(edtPageNumber.Text)-1].TabVisible:=False;
  if cbTabVisible.Checked
    then PageControl1.Pages[StrToInt(edtPageNumber.Text)-1].Visible:=True
    else PageControl1.Pages[StrToInt(edtPageNumber.Text)-1].Visible:=False;
end;

procedure TForm17.btnExitClick(Sender: TObject);
begin
  Close;
end;

procedure TForm17.FormDestroy(Sender: TObject);
begin
  TS.Free;
end;
end.
```

Флажки `cbVisible` и `cbTabVisible` задают параметры видимости страницы и ее содержимого.

Добавление к блокноту новой страницы в процедурах `FormCreate` и `btnAddPageClick` выполнено двумя способами: с использованием специальной переменной типа `TTabSheet` и без нее.

В программе не предусмотрена обработка исключений, поэтому возникающие ошибки будут обрабатываться глобальным обработчиком приложения. Исключение может возникнуть, например, при преобразовании в число строки `edtPageNumber.Text`.



ГЛАВА 9

Организация приложений

Delphi 7 позволяет разрабатывать приложения, функционирующие под управлением операционных систем Windows и Linux. В этом случае используется библиотека CLX (межплатформенный вариант библиотеки VCL). Для создания приложений на основе CLX на странице **New** в Хранилище объектов имеется объект `CLX Application`. Подробное рассмотрение вопросов, связанных с разработкой межплатформенных приложений, выходит за рамки этой книги, поэтому далее везде будут предполагаться только приложения под Windows (на основе VCL).

У каждого приложения есть одна главная форма. Кроме того, приложение может содержать различное количество других форм, например, диалоговых окон для установки параметров программы. При выполнении программы отдельные окна могут быть видимыми или невидимыми на экране. Созданием и удалением, отображением и скрытием окон, а также организацией взаимодействия между окнами управляет программист. Для этого используются соответствующие свойства, методы и события формы.

Для управления формами приложения также используются глобальные объекты `Application` (приложение) и `Screen` (экранная среда), уже рассмотренные в *главе 6*.

Как уже говорилось, в Windows различают два основных типа приложений: однодокументные (SDI) и многодокументные (MDI). Под *документом* понимается центральный объект, с которым работает приложение. Так, для системы Delphi документом является проект, для текстового процессора — содержимое текстового файла, для графического редактора — изображение из файла соответствующего графического формата. В принципе, документ не является однородным — он может содержать другие объекты. Например, в текстовый документ формата DOC смогут быть внедрены различные OLE-объекты (таблицы, рисунки и т. д.).

Однодокументное приложение в текущий момент времени может работать только с одним документом (центральным объектом). Примерами таких приложений служат Блокнот (Notepad) и графический редактор Paint. Хотя однодокументное приложение обрабатывает один документ, оно может содержать несколько окон: кроме главного окна, в процессе работы могут появляться и другие, например, диалоговые окна, плавающие панели инструментов и т. п. Характерным примером однодокументного приложения является сама интегрированная среда Delphi, которая часто отображает на экра-

не минимум четыре окна — главное окно, окно Конструктора формы, окно Редактора кода и окно Инспектора объектов. Можно открывать и другие окна. В однодокументном приложении каждое окно является самостоятельным и визуально отделено от других.

Отметим, что в приводимых в книге примерах, посвященных работе с формами и приложениями, речь идет именно об однодокументных приложениях.

Многодокументное приложение может работать одновременно с несколькими различными документами (центральными объектами). Характерными примерами многодокументных приложений являются текстовый и табличный процессоры Microsoft Word и Excel, графический редактор CorelDraw.

В многодокументном приложении главное окно содержит дочерние окна, размещаемые в его пределах. В дочерних окнах обычно выполняются просмотр и редактирование документов.

Создание многодокументных приложений

Общие принципы организации и взаимодействия форм многодокументного приложения те же, что и для однодокументного. Определенные особенности связаны с размещением элементов управления в главном окне, созданием и удалением дочерних окон, а также с управлением открытыми дочерними окнами.

Особенности многодокументных приложений

Многодокументное приложение имеет одну главную форму, внутри которой может размещаться несколько дочерних окон. Вид окна (главное или дочернее) определяется свойством `FormStyle`. Для главной формы приложения, и только для нее свойство `FormStyle` должно иметь значение `fsMDIForm`, а для дочерних окон — значение `fsMDIChild`. Установка значения свойства `FormStyle` может выполняться как на этапе разработки, так и при выполнении приложения.

Главная форма многодокументного приложения обычно не содержит такие элементы управления, как надписи, кнопки и таблицы строк. Если в главной форме разместить элемент управления, например, кнопку `Button`, то она будет видна ("просвечивать") через открытое дочернее окно.

Пользовательский интерфейс главного окна составляют меню, панели инструментов и строка состояния. Вся оставшаяся клиентская область главного окна используется для размещения дочерних окон, внутри которых и находятся требуемые элементы управления.

В свою очередь, *дочернее окно (форма)* обычно не содержит видимых меню, панелей инструментов и строк состояния. Если в дочерней форме имеется компонент `MainMenu`, то при создании формы это меню соединяется с меню главной формы. Так как соединение меню `MainMenu` главной и дочерней форм выполняется автоматически, свойство `AutoMerge` обоих меню должно быть установлено в значение `False`.

Меню дочернего окна обычно содержит команды, специфичные для операций, связанных с этим окном и содержащимся в нем документом. (Организация и использование меню рассматриваются в *главе 5*.)

При разработке приложения описываются классы форм главного и одного дочернего окна. По умолчанию в проекте предусматривается автоматическое создание и отображение этих форм на экране после запуска приложения. Остальные дочерние окна (второе, третье и т. д.) должны динамически создаваться программным способом. В частности, так работает текстовый процессор Microsoft Word, когда непосредственно после его загрузки пользователю предоставляется пустой документ с именем Документ1. При открытии текстовых файлов для каждого из них создается отдельное окно.

Если автоматическое создание первого экземпляра дочернего окна при запуске программы не требуется, то из файла проекта нужно удалить соответствующую инструкцию. Это можно выполнить, изменив файл проекта вручную, но проще воспользоваться окном параметров проекта, перенеся дочернюю форму из списка автоматически создаваемых в список доступных форм. Таким образом, описание класса дочерней формы, выполненное при проектировании приложения, становится доступным, и в процессе выполнения программы можно динамически создавать требуемое число экземпляров дочерней формы, например, так:

```
fmChild := TfmChild.Create(Application);
fmChild.Caption := 'Новое дочернее окно';
```

Отметим, что, несмотря на то что дочерняя форма размещается в пределах главной, ее владельцем является не главная форма, а приложение, поэтому в качестве параметра метода `Create` указан глобальный объект `Application`. При завершении работы с программой незакрытая дочерняя форма, как и другие формы приложения, автоматически уничтожается и удаляется из памяти.

После создания новой формы ее имя (в примере `fmChild`) указывает на экземпляр последней созданной формы. Для *доступа ко всем дочерним формам* следует использовать свойство `MDIChildren[I: Integer]` типа `TForm` главной формы приложения. Это свойство является массивом, который содержит дочерние окна многодокументного приложения. Отсчет номеров окон начинается с нуля, нулевой номер имеет верхнее окно. При переразмещении или переупорядочивании окон на экране их позиции в массиве `MDIChildren` изменяются, отражая новое положение на экране. *Число дочерних окон* многодокументного приложения определяется свойством `MDIChildCount` типа `Integer`.

Например, в процедуре

```
procedure TfmMain.mnuNumberClick(Sender: TObject);
var n: integer;
begin
for n := 0 to fmMain.MDIChildCount – 1 do
    fmMain.MDIChildren[n].Caption := 'Окно номер ' + IntToStr(n);
end;
```

в заголовках дочерних окон приложения выводятся номера этих окон (из свойства `MDIChildren` главной формы `fmMain` приложения).

Для доступа к активному дочернему окну удобно использовать свойство `ActiveMDIChild` типа `TForm` главной формы. Это свойство определяет, какое дочернее окно находится в фокусе ввода, и его можно использовать, например, для закрытия окна.

Замечание

Если приложение не является многодокументным, то использование свойства `ActiveMDIChild` приведет к исключению.

В процедуре

```
procedure TfmMain.mnuCloseChildClick(Sender: TObject);
begin
  if fmMain.ActiveMDIChild <> nil then fmMain.ActiveMDIChild.Close;
end;
```

перед закрытием формы проверяется ее существование.

Свойства `MDIChildren`, `MDIChildCount` и `ActiveMDIChild` доступны для чтения и действуют во время выполнения программы.

Закрытие главного окна многодокументного приложения, как и в однодокументном приложении, приводит к завершению работы всего приложения. При попытке закрытия дочернего окна одним из следующих способов:

- нажатием кнопки закрытия окна в области заголовка;
- нажатием комбинации клавиш <Ctrl>+<F4>;
- вызовом метода `Close` дочерней формы

по умолчанию происходит не закрытие, а *свертывание* этого окна. Чтобы окно закрывалось привычным для пользователя способом, следует создать обработчик события `OnClose` дочерней формы:

```
procedure TfmChild.FormClose(Sender: TObject;  var Action: TCloseAction);
begin
  Action := caFree;
end;
```

По умолчанию для дочерних окон параметр-признак `Action` имеет значение `caMinimize`, и вместо закрытия дочерняя форма сворачивается.

При программном закрытии дочернего окна с помощью метода `Close` во избежание ошибок следует выполнить предварительную проверку удаляемой формы на существование, например, как в предыдущем примере.

Замечание

Закрыть дочернее окно методом `Hide` нельзя: вызов этого метода приводит к возникновению исключения.

Если в дочерних формах производится редактирование документов, то при закрытии этих окон программист должен предусмотреть проверку того, сохранен ли соответствующий документ. Для этого можно использовать события `OnClose` и `OnCloseQuery`. Обработчик события `OnClose` будет, например, таким:

```
procedure TfmChild.FormClose(Sender: TObject;
          var Action: TCloseAction);
begin
  if Memo1.Modified then
    if MessageDlg('Файл не сохранен!' + #10#13' Подтверждаете выход?',
                  mnConfirmation, [mbYes, mbNo], 0) = mrYes
```

```
        then Action := caFree
        else Action := caNone;
end;
```

Если приложение использует дочерние окна нескольких типов, то на этапе проектирования для каждой формы нужно подготовить описание класса. Окна различных типов могут понадобиться, например, при составлении программы просмотра текстовых или графических файлов.

Для управления дочерними окнами у главной формы есть несколько методов, с помощью которых можно упорядочивать окна и перемещаться между ними.

Методы `Cascade` и `Tile` упорядочивают дочерние окна, располагая их *каскадом* (рис. 9.1, слева) или *мозаикой* (рис. 9.1, справа) соответственно.

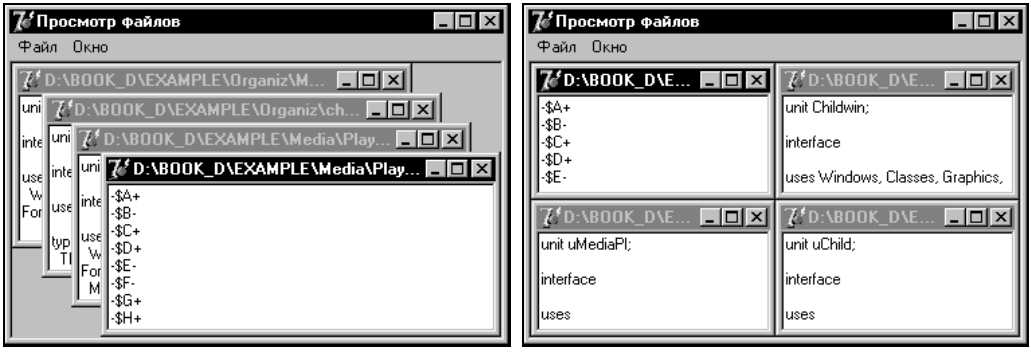


Рис. 9.1. Размещение дочерних окон каскадом и мозаикой

Замечание

В однодокументном приложении методы `Cascade` и `Tile` не действуют.

При размещении каскадом, если размеры главного окна недостаточно велики, чтобы отобразить дочерние окна так, как показано на рисунке, часть окон располагается поверх других окон, закрывая их заголовки. По умолчанию новые дочерние окна располагаются каскадом.

При упорядочивании окон мозаикой вариант их размещения определяется свойством `TileMode` типа `TTileMode`, принимающим следующие значения:

- `tbHorizontal` (дочернее окно располагается во всю ширину клиентской области родительской формы) — по умолчанию (рис. 9.2, слева);
- `tbVertical` (дочернее окно располагается во всю высоту клиентской области родительской формы; рис. 9.2, справа).

Замечание

Действие свойства `TileMode` имеет исключения. Так, если главная форма содержит четыре дочерних окна, то, независимо от значения свойства `TileMode`, размещены они будут так, как показано на рис. 9.1, справа.

Рис. 9.2. Варианты размещения окон мозаикой

Для *программного перемещения* между окнами можно использовать методы `Next` и `Previous`, которые передают фокус ввода и делают активным следующее или предыдущее дочернее окно соответственно. Пользователь может переключаться между окнами с помощью мыши или комбинаций клавиш <Ctrl>+<Tab> и <Ctrl>+<Shift>+<Tab>.

Кроме метода, выполняющего размещение раскрытых дочерних окон, у главной формы есть метод `ArrangeIcons`, предназначенный для *упорядочивания значков* свернутых (минимизированных) окон.

Замечание

На несвернутые окна метод `ArrangeIcons` не действует.

У многих многодокументных приложений в главном меню есть пункт **Окно**, подменю которого обычно содержит команды, предназначенные для манипулирования дочерними окнами. Состав этого подменю может быть, например, таким:

- **Каскад**;
- **Мозаика**;
- **Следующее окно**;
- **Предыдущее окно**.

При выборе одного из пунктов вызывается соответствующий метод, например, `Cascade` или `Next`. Формируется это подменю обычным способом, т. е. путем создания пунктов меню и обработчиков события их выбора.

Кроме того, в подменю пункта **Окно** главного меню часто содержится список раскрытых дочерних окон, в котором активное окно отмечено галочкой (рис. 9.3). Для *автоматического отображения* такого *списка* используется свойство `WindowMenu` типа `TMenuItem`. В качестве пункта меню можно указывать только пункт верхнего уровня, т. е. пункт, название которого отображается непосредственно в строке меню.

Так, в примере

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Эти операции проще выполнить через Инспектор объектов
  mnuWindow.Caption := '&Окно';
  Form1.WindowMenu := mnuWindow;
end;
```

подменю **Окно** главного меню формы `Form1` автоматически отображает список дочерних окон.

После установки значения свойства `WindowMenu` главной формы Delphi автоматически отслеживает открытие и закрытие дочерних окон, а также передачу фокуса ввода между ними, отражая в списке указанного пункта меню произошедшие изменения. Сделать окно активным можно щелчком на названии этого окна в списке.

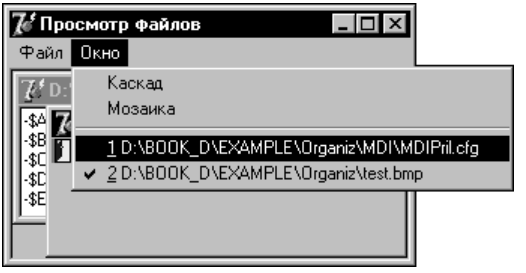


Рис. 9.3. Подменю **Окно** главного меню со списком открытых дочерних окон

Чтобы подменю **Окно** работало корректно, требуется программно анализировать наличие или отсутствие открытых дочерних окон и контролировать их количество, в зависимости от этого блокируя или разблокируя соответствующие пункты меню. Например, если нет открытых окон или открыто одно окно, то метод `Cascade` вызывать бесполезно, и команда **Каскад** должна быть неактивной.

Большинство событий формы для многодокументного приложения генерируется и используется обычным способом. Исключением является событие `OnActivate`, генерируемое двумя способами. Когда приложение активно, при переключении между его дочерними окнами событие `OnActivate` возникает для окна, которое получает фокус ввода. Если приложение было неактивно, то в момент его активизации событие `OnActivate` возникает для главной формы.

Пример многодокументного приложения

В качестве примера многодокументного приложения рассмотрим программу, предназначенную для просмотра текстовых и графических файлов. В главной форме приложения (рис. 9.4) находятся компоненты `MainMenu1`, `OpenDialog1` и `OpenPictureDialog1`.

Рис. 9.4. Главная форма многодокументного приложения

Меню содержит команды, которые позволяют управлять дочерними окнами, а также осуществить выход из программы. Структура меню следующая:

- **Файл** (`mnuFile`):
  - **Открыть текст** (`mnuOpenText`);
  - **Открыть графику** (`mnuOpenGraphic`);
  - **Закрыть окно** (`mnuCloseWindow`);
  - **Закрыть все** (`mnuCloseAll`);
  - **Выход** (`mnuExit`);
- **Окно** (`mnuWindow`):
  - **Каскад** (`mnuCascade`);
  - **Мозаика** (`mnuTile`);

В структуре меню указаны заголовки пунктов (свойство `Caption`), в скобках приведены имена пунктов (свойство `Name`). Разделитель, отделяющий пункт **Выход** от других пунктов, не указан. В заголовках пунктов комбинации клавиш не задаются.

Помимо главной формы, приложение содержит два вида дочерних окон, предназначенных для просмотра текстовых (рис. 9.5, слева) и графических (рис. 9.5, справа) файлов.

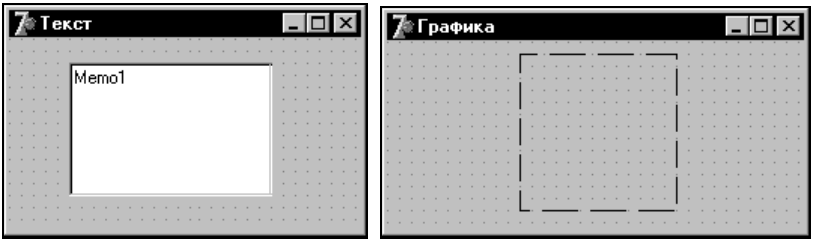


Рис. 9.5. Вид дочерних форм многодокументного приложения на этапе проектирования

В листинге 9.1 приведены тексты файла проекта и файлов модулей всех трех форм.

Листинг 9.1. Пример многодокументного приложения

```
// Файл проекта
program MDIApplication;
uses
  Forms,
  uMDIApplication in 'uMDIApplication.pas' {fmMain},
  uChildT in 'uChildT.pas' {fmChildText},
  uChildG in 'uChildG.pas' {fmChildGraphic};
{$R *.RES}
begin
  Application.Initialize;
  // Автоматически создается только главная форма приложения
  Application.CreateForm(TfmMain, fmMain);
```

```
  Application.Run;
end.

// Модуль главной формы приложения
unit uMDIApplication;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, ExtDlgs;
type
  TfmMain = class(TForm)
         MainMenu1: TMainMenu;
           mnuFile: TMenuItem;
       mnuOpenText: TMenuItem;
    mnuOpenGraphic: TMenuItem;
    mnuCloseWindow: TMenuItem;
       mnuCloseAll: TMenuItem;
                N1: TMenuItem;
           mnuExit: TMenuItem;
         mnuWindow: TMenuItem;
        mnuCascade: TMenuItem;
           mnuTile: TMenuItem;
       OpenDialog1: TOpenDialog;
OpenPictureDialog1: TOpenPictureDialog;

    procedure FormCreate(Sender: TObject);
    procedure mnuOpenTextClick(Sender: TObject);
    procedure mnuOpenGraphicClick(Sender: TObject);
    procedure mnuCloseWindowClick(Sender: TObject);
    procedure mnuCloseAllClick(Sender: TObject);
    procedure mnuCascadeClick(Sender: TObject);
    procedure mnuTileClick(Sender: TObject);
    procedure mnuExitClick(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  fmMain: TfmMain;
implementation
// Описание может добавляться автоматически при трансляции
uses uChildT, uChildG;
{$R *.DFM}

procedure TfmMain.FormCreate(Sender: TObject);
begin
  fmMain.Caption    := 'Просмотр файлов';
  fmMain.FormStyle  := fsMDIForm;
```

```
  fmMain.WindowMenu := mnuWindow;
end;

procedure TfmMain.mnuOpenTextClick(Sender: TObject);
begin
  // Создание экземпляра формы fmChildText
  // и загрузка выбранного текстового файла
  if OpenDialog1.Execute then begin
    fmChildText := TfmChildText.Create(Application);
    fmChildText.Caption := OpenDialog1.FileName;
    fmChildText.Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
  end;
end;

// Создание экземпляра формы fmChildGraphic
// и загрузка выбранного графического файла
procedure TfmMain.mnuOpenGraphicClick(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then begin
    fmChildGraphic := TfmChildGraphic.Create(Application);
    fmChildGraphic.Caption := OpenPictureDialog1.FileName;
    fmChildGraphic.Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
  end;
end;

procedure TfmMain.mnuCloseWindowClick(Sender: TObject);
begin
  if fmMain.ActiveMDIChild <> nil then fmMain.ActiveMDIChild.Close;
end;

procedure TfmMain.mnuCloseAllClick(Sender: TObject);
var n: integer;
begin
  for n := fmMain.MDIChildCount - 1 downto 0 do
    if fmMain.ActiveMDIChild <> nil then fmMain.MDIChildren[n].Close;
end;

procedure TfmMain.mnuExitClick(Sender: TObject);
begin
  Close;
end;

procedure TfmMain.mnuCascadeClick(Sender: TObject);
begin
  fmMain.Cascade;
end;

procedure TfmMain.mnuTileClick(Sender: TObject);
begin
  fmMain.Tile;
end;
end.
```

```
// Модуль формы для просмотра текстов
unit uChildT;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TfmChildText = class(TForm)
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  fmChildText: TfmChildText;
implementation
{$R *.DFM}

procedure TfmChildText.FormCreate(Sender: TObject);
begin
  FormStyle   := fsMDIChild;
  Memo1.Align := alClient;
end;

procedure TfmChildText.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  // Удаление формы при ее закрытии
  Action := caFree;
end;
end.

// Модуль формы для просмотра графики
unit uChildG;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls;
type
  TfmChildGraphic = class(TForm)
    Image1: TImage;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  fmChildGraphic: TfmChildGraphic;
implementation
{$R *.DFM}

procedure TfmChildGraphic.FormCreate(Sender: TObject);
begin
  FormStyle := fsMDIChild;
  Image1.AutoSize := True;
end;

procedure TfmChildGraphic.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  // Удаление формы при ее закрытии
  Action := caFree;
end;
end.
```

Для отображения информации каждая дочерняя форма содержит соответствующий элемент управления.

В форме `fmChildT` размещен многострочный редактор `Memo1`, внутри которого выводится содержимое текстового файла. При создании формы компонент `Memo1` выравнивается по всему размеру формы. Так как вывод текста выполняется в многострочном редакторе, с помощью данного приложения можно просматривать обычные текстовые файлы (ANSI, ASCII).

В форме `fmChildG` размещен компонент `Image1`, внутри которого отображается выбранный графический файл. При создании формы этот компонент принимает размеры согласно загруженному в него рисунку. Отображаться могут файлы форматов BMP, ICO, WMF.

Для всех форм установка свойства `FormStyle` в значения `fsMDIForm` и `fsMDIChild` выполнена при создании форм. Эти действия, а также задание значений свойств компонентов `Memo1` и `Image1` проще выполнить через Инспектор объектов на этапе разработки программы.

Автоматическое создание дочерних форм не требуется, поэтому из проекта исключены соответствующие инструкции, относящиеся к созданию форм `fmChildT` и `fmChildG`.

В приведенном приложении вместо двух типов дочерних окон (для просмотра текстовых и графических файлов соответственно) можно использовать один тип, общий для всех файлов. Возможен вариант, когда в этой дочерней форме размещаются оба компонента: `Memo` и `Image`. Открываемый файл в зависимости от содержащейся в нем информации загружается в нужный компонент, а второй компонент делается невидимым. Другой вариант предусматривает динамическое создание компонента `Memo` или `Image`.

Два вида дочерних окон созданы независимо друг от друга на базе класса `TForm`. Можно *создать одно окно на основе другого*, например, форму для отображения графики на основе формы для отображения текстов. При этом описание типа формы может выглядеть так:

```
uses uChildT;
type
  TfmChildGraphic = class(TfmChildText)
  ...
  end;
```

Так как базовый класс `TfmChildText` включает компонент `Memo1`, который при выводе графики не нужен, при выполнении программы его желательно *динамически удалить*. Это можно выполнить при создании формы `fmChildGraphic`.

```
procedure TfmChildGraphic.FormCreate(Sender: TObject);
begin
  inherited;
  Memo1.Free;
end;
```

Добавление компонента `Image1` к форме выполняется обычным способом при разработке приложения, можно также создать его динамически.

При выборе имени *файла для открытия* можно использовать стандартное диалоговое окно `OpenDialig1`. Для удобства пользователя можно настроить это диалоговое окно на открытие графических файлов, установив соответствующее значение свойства `Filter`.

В приведенном примере для простоты не выполняется блокировка отдельных пунктов меню, которую желательно предусматривать в реальных приложениях. Например, при отсутствии дочерних окон следует блокировать пункты меню **Закрыть окно** и **Каскад**.

В заголовках дочерних окон выводятся полные имена просматриваемых файлов. Можно отображать только имя файла и его расширение, которые выделяются с помощью функции `ExtractFileName`.

Шаблон многодокументного приложения

Для быстрого создания проекта многодокументного приложения можно воспользоваться Хранилищем объектов и выбрать на его странице **Projects** шаблон **MDI**

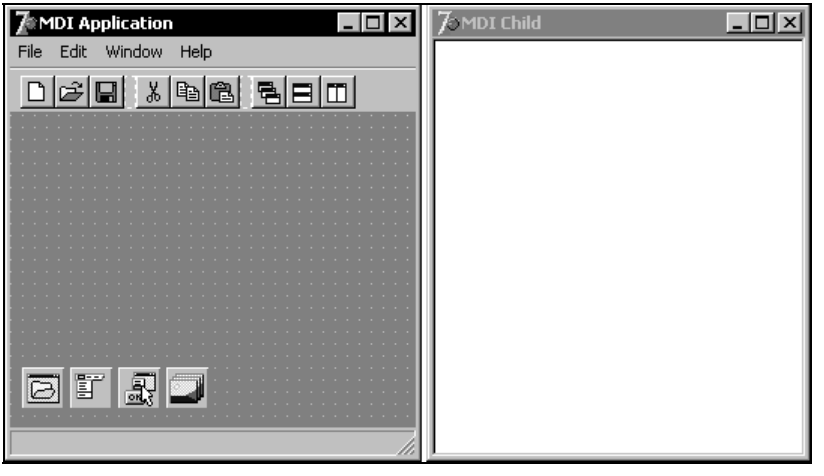


Рис. 9.6. Главная и дочерняя формы шаблона приложения

Application. При этом создается новый проект, включающий главную форму (рис. 9.6, слева), дочернюю форму (рис. 9.6, справа) и информационное окно.

Такой шаблон хорошо подходит для создания многодокументных текстовых редакторов. Формы приложения уже имеют соответствующие элементы управления. Так, главная форма содержит меню, панель инструментов, строку состояния, стандартный диалог выбора имени файла для открытия, список действий и список графических изображений. В дочернюю форму включен многострочный редактор. Кроме компонентов, обе формы содержат код организации взаимодействия между формами, выполняющий такие действия, как открытие дочерней формы и загрузка в нее текстового файла, а также корректное закрытие дочерней формы.

Информационное окно представляет собой диалоговое окно, обычно имеющее заголовок **О программе**.

Вывод заставки

Обычно программа в начале работы выводит на экран окно (панель) с краткой информацией о себе. Это окно присутствует на экране в процессе загрузки приложения, после чего исчезает автоматически или по команде пользователя. Подобное информационное окно также называют *заставкой*.

Для отображения заставки можно использовать различные приемы. Рассмотрим случай, когда заставка представляет собой специально предназначенную для этого *форму*. При запуске приложения форма-заставка создается и отображается первой. Далее создается главная форма приложения, в которой выполняется задержка на время отображения заставки на экране. По истечении этого времени заставка удаляется с экрана и из памяти и продолжается обычное выполнение приложения.

Для отображения заставки описанным способом необходимо внести определенные изменения в файлы проекта и модуля главной формы.

Рассмотрим следующий пример. Пусть приложение `InfoDemo` имеет в своем составе две формы: главную форму приложения `fmMain` (рис. 9.7, справа) и форму заставки `fmInfo` (рис. 9.7, слева).

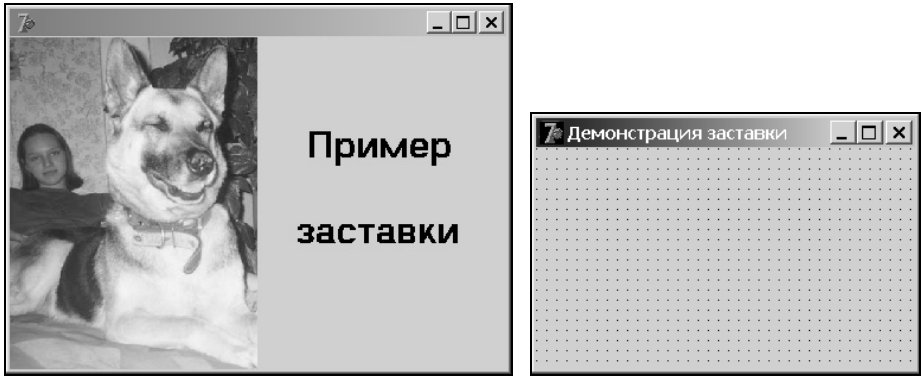


Рис. 9.7. Вид заставки и главной формы во время разработки приложения

В листинге приведены тексты файлов проекта, модулей главной формы и заставки: InfoDemo.dpr, Main.pas и Info.pas.

Листинг 9.2. Пример вывода заставки

```
// Файл проекта InfoDemo.dpr
program InfoDemo;
uses
  Forms,
  Main in 'Main.pas' {fmMain},
  Info in 'Info.pas' {fmInfo};
{$R *.RES}
begin
  Application.Initialize;
  // Создание и отображение формы-заставки
  fmInfo := TfmInfo.Create(Application);
  fmInfo.Show; fmInfo.Update;
  // Создание главной формы приложения
  Application.CreateForm(TfmMain, fmMain);
  // Удаление формы-заставки
  fmInfo.Hide; fmInfo.Free;
  Application.Run;
end.

// Файл Main.pas модуля главной формы fmMain
unit Main;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs;
type
  TfmMain = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  fmMain: TfmMain;
implementation
uses zast;
{$R *.DFM}

procedure TfmMain.FormCreate(Sender: TObject);
var t: longint;
begin
  t := GetTickCount div 1000;
  while (GetTickCount div 1000) < t + 5 do;
end;
end.
```

```
// Файл Info.pas модуля формы-заставки fmInfo
unit Info;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls;
type
  TfmZast = class(TForm)
    Image1: TImage;
     Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  fmZast: TfmZast;
implementation
{$R *.DFM}

// Все установки, сделанные в этой процедуре,
// проще выполнить с помощью Инспектора объектов
procedure TfmInfo.FormCreate(Sender: TObject);
begin
  // Установка параметров формы-заставки
  fmInfo.Position    := poScreenCenter;
  fmInfo.FormStyle   := fsStayOnTop;
  fmInfo.Caption     := '';
  fmInfo.BorderStyle := bsNone;
  fmInfo.BorderIcons := [];
  // Установка параметров текстового поля
   Memo1.BorderStyle := bsNone;
  Memo1.Align       := alRight;
  Memo1.Width       :=  fmInfo.ClientWidth div 2;
  Memo1.Color       := clGreen;
  Memo1.Lines.Clear;
  Memo1.Lines.Add('');
  Memo1.Lines.Add('');
  Memo1.Lines.Add('Пример');
  Memo1.Lines.Add('');
  Memo1.Lines.Add('заставки');
  Memo1.Lines.Add('');
  Memo1.Font.Color := clPurple;
  Memo1.Font.Size := 14;
  Memo1.Alignment := taCenter;
  // Установка параметров рисунка
  Image1.Align   := alLeft;
  Image1.Width   := fmInfo.ClientWidth div 2;
  Image1.Stretch := True;
end;
end.
```

Создание, отображение и удаление формы-заставки производятся в файле проекта. Сразу после инициализации приложения создается экземпляр формы `fmInfo`:

```
fmInfo := TfmInfo.Create(Application);
```

Так как создание формы не приводит к ее автоматическому появлению на экране, необходимы инструкции, отображающие заставку:

```
fmInfo.Show; fmInfo.Update;
```

Обычно для *отображения формы* достаточно метода `Show`. В данной ситуации дополнительно следует вызвать метод `Update`, т. к. объект-приложение еще не запущен и, соответственно, не работает цикл обработки сообщений Windows. Поэтому сообщения прорисовки формы автоматически не обрабатываются, и нужно выполнить отображение формы самостоятельно с помощью метода `Update`.

После создания и отображения заставки обычным способом создается главная форма приложения `fmMain`. В обработчике события `OnCreate` главной формы отсчитывается время (5 с) показа заставки на экране:

```
t := GetTickCount div 1000;
while (GetTickCount div 1000) < t + 5 do;
```

Если при создании главной формы в обработчике `FormCreate` выполняется много действий, то задержка может не потребоваться. Но в этом случае время отображения заставки на экране будет зависеть от производительности компьютера.

По истечении требуемого времени задержки форма-заставка делается невидимой и удаляется:

```
fmInfo.Hide; fmInfo.Free;
```

Задержка на заданное время может не понадобиться и когда в состав приложения входит много форм, автоматическое создание которых выполняется в файле проекта. В этом случае инструкции создания форм проекта располагаются между инструкциями, которые создают и удаляют форму-заставку. В результате панель заставки отображается сразу после запуска программы и видна в течение времени, необходимого на подготовку других форм приложения.

Заставка представляет собой форму, не имеющую заголовка и расположенную поверх всех окон (стиль формы `fsStayOnTop`). В форме заставки расположены два компонента: `Image` и `Memo`. Графическое изображение `Image1` содержит рисунок, загружаемый при разработке приложения, а многострочный редактор `Memo1` — информационный текст `Пример заставки`. В приведенном примере значения свойств формы и ее компонентов задаются при создании заставки, на практике более удобно выполнять установки, содержащиеся в обработчике `TfmInfo.FormCreate`, с помощью Инспектора объектов.

Замечание

Если содержащийся в компоненте `Image` рисунок имеет значительный объем, то предпочтительнее загружать его при выполнении приложения, а не включать в исполняемый файл программы, чтобы не увеличивать размер этого файла.

Пример загрузки рисунка при выполнении приложения:

```
// Файл с рисунком заставки должен находиться в папке
// с исполняемым файлом приложения.
// В случае ошибки поиска или загрузки рисунка сообщение об ошибке
// не выдается.
try
  Image1.Picture.LoadFromFile('Dog.bmp');
  except
end;
```

Другим вариантом вывода заставки является отображение в качестве информационной панели окна **О программе**. Здесь разработчик создает форму-заставку обычным способом, а ее вывод на экран осуществляется в модальном режиме. В этом случае файл проекта может иметь следующий вид:

```
program InfoDemo2;
uses
  Forms,
  uInfo2 in 'uZast2.pas' {fmMain2},
  u2Info2 in 'u2Zast2.pas' {fmInfo2};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TfmMain2, fmMain2);
  Application.CreateForm(TfmInfo2, fmInfo2);
  // Вывод формы-заставки в модальном режиме
  fmInfo2.ShowModal;
  // Удаление формы-заставки
  fmInfo2.Free;
  Application.Run;
end.
```

В приведенной программе `fmMain2` — главная форма приложения, а `fmInfo2` — заставка. После закрытия заставки экземпляр ее формы удаляется из памяти и при необходимости повторного использования форма снова должна быть создана конструктором `Create`.

Вывод информационного окна

Большинство приложений способно отображать *информационное окно*, в котором выводится краткая информация о приложении. Обычно это сведения о названии программного продукта, его версии, дате выпуска, авторах, а также другие данные и небольшой рисунок. Информационное окно часто имеет заголовок **О программе** (About).

Создать такое окно можно на основе *шаблона* или самостоятельно.

При использовании шаблона в Хранилище объектов на странице **Forms** (Формы) нужно выбрать шаблон формы **About box** (Информационное окно). В результате к приложению добавляется форма с именем `AboutBox` и заголовком **About** (рис. 9.8).

Рис. 9.8. Шаблон информационного окна **About**

В форме расположены информационная панель и кнопка **OK** закрытия окна. Свойство `ModalResult` кнопки установлено в значение `mrOK`, поэтому при ее нажатии форма `AboutBox` автоматически закрывается. При этом обработчик события нажатия кнопки не требуется. На панели находятся четыре надписи `Label` и компонент-изображение `Image`. Для отображения в окне сведений о программном продукте и его разработчиках необходимо изменить значения свойств `Caption` компонентов `Label`. В компонент `Image` можно загрузить рисунок, поясняющий назначение программы. Заголовок окна также можно изменить.

При необходимости можно программным путем удалить из формы ненужные элементы управления или добавить какие-либо элементы, установив для них соответствующие значения свойств.

Как правило, операции с информационным окном и его элементами выполняются при разработке программы с помощью Инспектора объектов. При этом события элементов не используются и обработчики событий не создаются.

Вместо выбора информационного окна на основе шаблона разработчик может *создать окно самостоятельно*. Для этого к проекту добавляется новая форма, в которой размещаются требуемые элементы управления.

При выполнении программы вывод информационного окна на экран обычно осуществляется в модальном режиме. Для этого используется метод `ShowModal`. Для продолжения работы с приложением информационное окно должно быть закрыто. Вызов окна **О программе** чаще всего располагается в обработчике событий пункта **О программе** или **About** главного меню.

Так, в следующем примере модальное окно `AboutBox` выводится на экран при выборе пункта меню `mnuAbout`:

```
procedure TForm1.mnuAboutClick(Sender: TObject);
begin
  AboutBox.ShowModal;
end;
```

Предполагается, что форма `AboutBox` уже создана, скажем, при запуске приложения. При вызове метода `ShowModal` информационное окно отображается на экране, а после его закрытия становится невидимым. Возможен вариант, когда форма `AboutBox` дина-

мически создается перед ее отображением, а после закрытия удаляется из памяти, как, например, в следующей процедуре:

```
procedure TForm1.mnuAboutClick(Sender: TObject);
begin
  AboutBox := TAboutBox.Create(Application);
  AboutBox.ShowModal;
  AboutBox.Free;
end;
```

Такой вариант позволяет ускорить загрузку приложения, т. к. на создание информационной формы затрачивается определенное время. Кроме того, форма `AboutBox` требуется пользователю достаточно редко, а остальное время работы программы она только занимает лишние ресурсы. При работе на современных компьютерах проблема недостатка ресурсов обычно не так остра, однако следование правилу "не потреблять лишние ресурсы" — признак хорошего стиля программирования.

Создание одноэкземплярного приложения

Некоторые приложения должны запускаться только *в одном экземпляре*. То есть пользователь не имеет возможности загрузить вторую копию этого приложения при работающей первой. Один из вариантов реализации такого одноэкземплярного приложения заключается в том, что программа перед своим запуском просматривает заголовки уже открытых окон и проверяет, не открыто ли ее главное окно.

С этой целью можно использовать API-функцию `FindWindow(lpClassName, lpWindowName: LPCTSTR): HWND`, которая возвращает дескриптор найденного окна. Параметр `lpClassName` задает ссылку на класс окна, а параметр `lpWindowName` определяет заголовок окна этого класса. Если требуемое окно отсутствует, то функция возвращает ноль, а в случае успешного поиска — дескриптор найденного окна.

При создании экземпляра окна Windows регистрирует имя класса окна. Delphi в качестве такого класса использует класс формы, т. е. когда Delphi создает экземпляр `Form1` класса `TForm1`, то `TForm1` применяется в качестве имени для регистрации окна `Form1`. Кроме того, каждая форма имеет заголовок, определяемый ее свойством `Caption`. Таким образом, задав в качестве параметров название класса и заголовок формы, с помощью функции `FindWindow` можно определить, зарегистрирована ли эта форма как окно Windows.

При создании одноэкземплярного приложения эту функцию нельзя вызывать из класса `TForm1`, т. к. к моменту выполнения кода модуля, содержащего описание класса `TForm1`, окно уже создано и будет найдено при поиске. Поэтому код, выполняющий поиск, должен находиться только в файле проекта:

```
program Appl;
uses
  Forms,
  Windows,       // Не забудьте подключить этот модуль!
  uAppl in 'uAppl.pas' {Form1};
```

```
{$R *.RES}
begin
  Application.Initialize;
  // Если окно уже существует, то завершить приложение
  if  FindWindow('TForm1', 'Form1') <> 0 then Application.Terminate;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

В этом примере главной формой приложения является форма `Form1` класса `TForm1`. Если эта форма уже существует в системе, то приложение завершает работу. Так как в проекте используется API-функция Windows, то в раздел `uses` проекта должен быть включен модуль `Windows`.

Следует иметь в виду, что при изменении свойств `Name` и `Caption` главной формы приложения нужно внести соответствующие коррективы и в файл проекта. Кроме того, неприятности может причинить возможное совпадение значений используемых свойств формы в разрабатываемом приложении и каком-либо другом, что приведет к невозможности запуска программы.

Иным подходом к созданию одноэкземплярного приложения является не просто блокировка запуска второй копии программы, а *активизация уже запущенного приложения*. Для этого можно использовать еще одну API-функцию, а именно `SetForegroundWindow(hwndWindow: HWND): boolean`. Эта функция размещает поверх всех открытых на экране окон окно, дескриптор которого задан параметром `hwndWindow`, и возвращает результат этой операции как логическое значение.

В этом случае проект приложения имеет следующий вид:

```
program Appl2;
uses
  Forms,
  Windows,
  uAppl2 in 'uAppl2.pas' {Form1};
{$R *.RES}
var hPrevWin: HWND;
begin
  Application.Initialize;
  hPrevWin := FindWindow('TForm1', 'Form1');
  if hPrevWin <> 0 then begin
     SetForegroundWindow(hPrevWin);
     Application.Terminate;
  end;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Если окно зарегистрировано и функция `FindWindow` в качестве ссылки на него возвращает ненулевое значение, запоминаемое в переменной `hPrevWin`, то окно с помощью функции `SetForegroundWindow` переводится на передний план.

Особенности консольного приложения

Консольным называется приложение, имитирующее работу в текстовом режиме экрана. Создание консольного приложения целесообразно, например, в случаях, когда к интерфейсной части приложения не предъявляются строгие требования, и пользователь работает с программой практически так же, как в среде DOS. При запуске консольного приложения Windows выделяет окно как для программы DOS, при этом в заголовке окна отображается название исполняемого файла приложения. Ввод/вывод данных осуществляется с помощью обычных процедур `read`, `readln`, `write` и `writeln`. К окну автоматически подключаются стандартные файлы `Input` и `Output`.

Замечание

Несмотря на то что работа пользователя с консольным приложением выглядит, как работа с программой DOS, это приложение является приложением Windows и в режиме DOS работать не будет.

Достоинством консольных приложений является относительная простота использования и переноса программ, написанных на языке Pascal, например на Borland Pascal 7.0, в систему программирования Delphi. Кроме того, исполняемый файл консольной программы намного меньше по размеру (десятки килобайт) по сравнению с исполняемым файлом Delphi-варианта такой же программы (сотни килобайт).

Для создания консольного приложения можно выбрать в Хранилище объектов на странице **New** объект **Console Wizard** (Мастер консольного приложения), в результате чего создается новый проект, состоящий из одного файла с расширением dpr. Этот файл и является консольной программой. Первоначально он выглядит так:

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

Приведенный код похож на заготовку обычной программы на языке Pascal, написанной в среде DOS. Единственным отличием является директива `$APPTYPE`, значение `CONSOLE` которой сообщает компилятору, что Delphi-программа работает в консольном режиме. Значение по умолчанию (`GUI`) соответствует созданию программы в среде Delphi.

Консольное приложение можно получить также на основе проекта обычного приложения следующим способом. Первоначально командой **File | New | Application** (Файл | Новый | Приложение) генерируется новое приложение, для которого также создаются файлы проекта. В составе консольного приложения нет форм, поэтому из проекта нужно удалить форму `Form1`. С этой целью командой **Project | Remove from Project** открывается диалоговое окно удаления форм из проекта (**Remove from Project**), в котором для данного проекта содержится один модуль `Unit1` формы. Нужно выбрать этот модуль и удалить без сохранения на диске.

После этого командой **Project | View Source** вызывается окно Редактора кода, и в нем открывается файл (dpr) проекта, который нужно модифицировать так, чтобы он принял приведенный выше вид.

Далее к заготовке консольного приложения добавляется код, определяющий собственно функциональность приложения.

Пример простейшей консольной программы:

```
program demo;
{$APPTYPE CONSOLE}
uses sysutils;
begin
  writeln('Простейшее консольное приложение');
  readln;
end.
```

В программе выполняется вывод текста `Простейшее консольное приложение`, после чего для продолжения дальнейшей работы нужно нажать клавишу <Enter>. При этом выполнение приложения прекращается, а окно автоматически закрывается.

Как и любая программа на языке Pascal, консольное приложение может включать отдельные модули, например, `unit1`. Подготовка и подключение модулей к основной программе выполняются как обычно.

При необходимости консольное приложение можно оттранслировать и отладить, не загружая среду Delphi. Для этого используется компилятор DCC32, вызываемый командой вида

```
dcc32.exe console.dpr
```

Команду можно задать, например, в диалоговом окне команды **Выполнить** главного меню Windows, а также в командной строке приложения Norton Commander или Windows Commander.

Исполняемый и конфигурационный файлы компилятора находятся в подкаталоге \BIN главного каталога Delphi.

В модуле `System` есть специальная логическая переменная `IsConsole`, с помощью которой можно определить режим выполнения приложения. Если переменная имеет значение `True`, то приложение является консольным, в противном случае — обычным приложением Delphi.

Консольное приложение Delphi представляет собой не просто программу, написанную на языке Delphi и выполняемую в среде Windows. Delphi поддерживает создание 32-разрядных консольных приложений, имеющих доступ к ресурсам системы и использующих различные API-функции Windows. При этом в разделе `uses` нужно указать модули, средства которых применяются в приложении.

Например, в программе

```
program Console;
uses Windows;
{$AppType Console}
```

```
begin
  SetConsoleTitle('Консольное приложение');
  readln;
end.
```

используется API-функция `SetConsoleTitle` для изменения заголовка окна на `Консольное приложение`. Эта функция входит в состав модуля `Windows`, поэтому имя модуля указано в разделе `uses`.

Запуск других приложений

В ряде случаев из приложения требуется *запускать* на выполнение другие приложения. Для этого можно использовать, например, API-функции `WinExec` и `ShellExecute`.

Проще всего использовать функцию `WinExec(lpCmdLine: LPCSTR, uCmdShow: UINT): UINT`, которая выполняет команду, заданную параметром `lpCmdLine`. В ней указывается имя исполняемого файла приложения, а также дополнительная информация (параметры программы). Если в имени исполняемого файла путь не указан, то поиск файла осуществляется в следующих каталогах:

- в каталоге, из которого запущено приложение;
- в текущем каталоге Windows;
- в системном каталоге Windows, название которого возвращает функция `GetSystemDirectory`;
- в главном каталоге Windows, название которого возвращает функция `GetWindowsDirectory`;
- в каталогах, заданных для поиска командой `PATH`.

С помощью параметра `uCmdShow` выбирается способ отображения окна запускаемого приложения. К важнейшим значениям параметра `uCmdShow` относятся:

- `SW_ShowMaximized` (окно развернуто);
- `SW_ShowMinimized` (окно свернуто);
- `SW_ShowNormal` (обычный вид окна).

После запуска новое приложение выполняется как обычно — вне зависимости от работы вызвавшего его приложения.

В качестве результата функция `WinExec` возвращает целое число, значение которого при успешном выполнении больше 31. При неудачном выполнении могут быть возвращены следующие значения:

- `0` (недостаточно памяти или других ресурсов);
- `ERROR_BAD_FORMAT` (неправильный формат exe-файла);
- `ERROR_FILE_NOT_FOUND` (не найден указанный файл);
- `ERROR_PATH_NOT_FOUND` (не найден указанный путь).

Замечание

При неудачном выполнении функции `WinExec` сообщения пользователю не выдаются.

Например, так можно вызвать калькулятор:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  WinExec(PChar('calc.exe'), SW_ShowNormal);
end;
```

Функция `WinExec` является 16-разрядной и оставлена для совместимости с более ранними версиями Delphi. Тем не менее, она позволяет запускать как 16-, так и 32-разрядные приложения Windows. При этом 16-разрядный код функции выполняется несколько медленнее его 32-разрядных аналогов, однако работа самого приложения, запущенного с помощью функции `WinExec`, от этого не зависит.

Функция `ShellExecute` является 32-разрядной и предоставляет больше возможностей, чем функция `WinExec`, но зато при ее вызове нужно задать много параметров.

Функция `ShellExecute(hwnd: HWND, lpOperation: LPCTSTR, lpFile: LPCTSTR, lpParameters: LPCTSTR, lpDirectory: LPCTSTR, nShowCmd: INT): HINSTANCE` позволяет *открыть* или *вывести* на принтер указанный *файл*, а также открыть *каталог*. В качестве файла (параметр `lpFile`) можно указать исполняемый файл, в результате чего запускается приложение, либо файл документа, что приводит к запуску текстового процессора (например, для текстовых документов с расширением doc будет запущен Microsoft Word).

Параметр `hwnd` содержит ссылку на окно, из которого запускается другое приложение.

Параметр `lpOperation` указывает на строку с описанием выполняемой команды (операции). Командная строка может принимать следующие значения:

- `open` (открытие файла или каталога, указанных параметром `lpFile`); если задан исполняемый файл приложения, то оно запускается;
- `print` (печать файла документа, указанного параметром `lpFile`);
- `explore` (открытие каталога, указанного параметром `lpFile`).

Если для параметра `lpOperation` указать значение `nil`, то по умолчанию принимается значение `open`.

Параметр `lpFile` является указателем на строку с именем файла или каталога, для которых выполняется операция.

Параметр `lpParameters` указывает на строку, содержащую передаваемые запускаемому приложению параметры. Если в качестве файла указывается неисполняемый файл, то параметру `lpParameters` следует задать значение `Nil`.

Параметр `lpDirectory` указывает на строку с именем каталога по умолчанию.

Параметр `nShowCmd` определяет способ отображения окна запускаемого приложения. Возможные значения этого параметра совпадают со значениями параметра `uCmdShow` функции `WinExec`.

В качестве результата функция `ShellExecute` возвращает ссылку на вновь запущенное приложение, значение которой в случае успешного выполнения больше 32. При неудачном выполнении функции возвращается код ошибки — число, меньшее или равное 32.

Перед использованием функции `ShellExecute` в разделе `uses` следует указать модуль `ShellAPI`.

В завершение раздела приведем несколько примеров, в которых для разных целей используются вызовы функции `ShellExecute`.

- Запуск текстового редактора Блокнот (Notepad):

```
uses ShellAPI;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShellExecute(Application.MainForm.Handle, Nil, PChar('notepad.exe'),
               Nil, Nil, SW_ ShowNormal);
end;
```

  Текстовый редактор Блокнот (Notepad) запускается из каталога Windows. В качестве окна, вызвавшего этот редактор, указана главная форма приложения `Application.MainForm`.

- Открытие текстового документа:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  ShellExecute(Application.MainForm.Handle, PChar('open'),
     PChar('test.doc'), Nil, PChar('d:\work'), SW_ ShowNormal);
end;
```

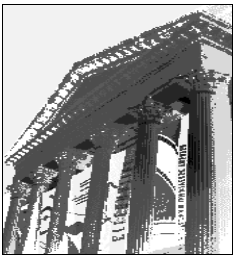
  Открывается файл test.doc, находящийся в каталоге d:\work. При этом автоматически запускается приложение, назначенное для обработки документов этого типа (обычно текстовый процессор Microsoft Word).

- Открытие папки:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  ShellExecute(Form1.Handle, PChar('open'), PChar('d:\examples'), Nil,
     Nil, SW_ ShowNormal);
end;
```

  Запускается Проводник Windows, в котором выводится содержимое каталога d:\examples.

ГЛАВА 10



Работа с графикой

Приложения Windows осуществляют вывод графической информации на экран или принтер с помощью функций GDI (Graphics Devices Interface — интерфейс графических устройств). Сама операционная система Windows является графической средой и для отображения информации также использует функции GDI.

Реализованные GDI-функции являются аппаратно-независимыми. Поэтому при выводе графической информации приложение работает не с физическим, а с логическим устройством, которое характеризуется широкой цветовой палитрой, высоким разрешением и т. п. Приложения взаимодействуют с устройствами вывода посредством драйверов, которые преобразуют аппаратно-независимые функции GDI в команды конкретного устройства.

При выполнении запроса приложения на вывод информации GDI или драйвер корректируют выводимую информацию с учетом ограниченных возможностей и особенностей физического устройства. Например, приложение может указать для цвета геометрической фигуры любой из примерно 16 миллионов цветов (2^{64}), однако далеко не любое физическое устройство (в частности, струйный принтер) обладает такими богатыми возможностями отображения цвета. Поэтому фигура будет окрашена в цвет, поддерживаемый конкретным устройством и наиболее близкий к запрошенному. Аналогичные преобразования выполняются для шрифта выводимых символов.

Такой подход позволяет приложению и операционной системе Windows функционировать относительно независимо от особенностей периферийного оборудования. Приложения Windows способны работать на компьютерах практически любой конфигурации, при этом чем лучше будут характеристики аппаратной части, тем больше качество выводимой информации будет соответствовать требованиям приложения.

Взаимодействие приложения с драйвером устройства осуществляется через специальную структуру данных, которая используется функциями GDI. Эта структура называется *контекстом отображения (дисплейным контекстом)* — DC (Display Context) и содержит основные характеристики устройства вывода, а также инструменты для рисования.

К контексту отображения относятся следующие три инструмента:

- шрифт;
- перо;
- кисть.

Схематично взаимодействие приложения и устройства вывода показано на рис. 10.1.

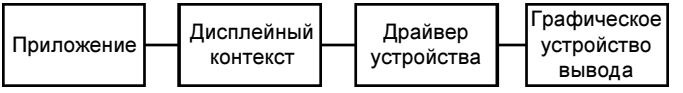


Рис. 10.1. Схема взаимодействия приложения и устройства вывода

Программирование графики в Windows — достаточно сложный и трудоемкий процесс, но Delphi и здесь приходит на помощь программисту и предлагает специальные классы, существенно упрощающие использование графических средств:

- `TCanvas` (для контекста отображения);
- `TFont` (для шрифта);
- `TPen` (для пера);
- `TBrush` (для кисти).

Связанные с этими классами объекты при необходимости создаются автоматически для всех визуальных компонентов (в том числе для формы). Поэтому у них есть свойства `Canvas`, `Font`, `Pen`, `Brush`. Однако перечисленные свойства доступны не у всех визуальных компонентов. Например, у формы `Form` класса `TForm` доступны свойства `Canvas`, `Font`, `Brush`, у кнопки `Button` класса `TButton` — `Font`, `Brush`, у геометрической фигуры `Shape` класса `TShape` — `Font`, `Pen`, `Brush`.

Кроме указанных выше, для работы с изображениями Delphi предлагает также классы:

- `TPicture` (контейнер для изображения);
- `TGraphic` (базовый класс для графических объектов-изображений);
- `TBitmap` (растровое изображение);
- `TIcon` (значок);
- `TMetaFile` (метафайл).

Эти классы инкапсулируются другими классами, например, `TImage`; экземпляры классов можно создавать и использовать программно.

Система Delphi предоставляет возможность рисовать на поверхности компонентов в процессе выполнения приложения и создавать изображения при конструировании приложения. Есть также возможность построения диаграмм.

Рисование при выполнении программы

Изображение можно строить при выполнении программы, например, рисуя на поверхности формы различными инструментами. В этом случае изображение представляет

собой комбинацию графических *примитивов* (простейших фигур), таких как точка, линия, круг или прямоугольник. Наряду с графическими примитивами возможен также вывод текста.

Графические операции часто используются для программной прорисовки видимой области компонентов. Различные визуальные компоненты способны сами отображать себя и свои элементы, что чаще всего и происходит. Однако некоторые компоненты, например, списки `ListBox` и `ComboBox`, строка состояния `StatusBar`, таблицы `DrawGrid` и `StringGrid`, многостраничный блокнот `PageControl` предоставляют также возможность отображать их видимую область или определенную ее часть программно. (Примеры программ, связанные с программной прорисовкой различных компонентов, приводятся в главах, посвященных этим компонентам.)

На *способ прорисовки* области компонента обычно указывает специальное свойство. Так, для компонента `PageControl` — это свойство `OwnerDraw`, а для компонента `ListBox` — свойство `Style`. Аналогичные свойства есть и у других элементов управления. Задав такому свойству компонента соответствующее значение, программист самостоятельно кодирует операции, связанные с выводом данных в области компонента. Обычно эти операции выполняются в процедуре — обработчике события, генерируемого при необходимости перерисовки компонента или его элемента. Например, для компонента `PageControl` — это событие `OnDrawTab`, а для компонента `ListBox` — событие `OnDrawItem`. В табл. 10.1 приведены свойства и события рассмотренных ранее компонентов, допускающих программную прорисовку своей области. Для свойств указаны значения, при которых выполняется обработчик события, связанного с прорисовкой области компонента.

Таблица 10.1. *Характеристики компонентов для программной прорисовки*

Компонент	Свойство	Значение	Событие
`ListBox`	`Style`	`lbOwnerDrawFixed` `lbOwnerDrawVariable`	`OnDrawItem`
`ComboBox`	`Style`	`csOwnerDrawFixed` `csOwnerDrawVariable`	`OnDrawItem`
`StringGrid`	`DefaultDrawing`	`False`	`OnDrawCell`
`PageControl`	`OwnerDraw`	`True`	`OnDrawTab`
`Panels` для `StatusBar`	`Style`	`psOwnerDraw`	`OnDrawPanel`

Замечание

Для большинства компонентов даже при наличии обработчика, связанного с программной прорисовкой, этот обработчик не вызывается, если соответствующее свойство компонента имеет значение, отличное от указанного в таблице.

Основной класс для связанных с рисованием графических операций — это `TCanvas`. С помощью его свойств и методов можно рисовать на поверхности визуальных объектов, которые включают в себя этот класс и, соответственно, имеют свойство `Canvas`.

К ним относятся, например, такие объекты, как форма Form, надпись Label, графическое изображение Image. Наиболее часто рисование производится на поверхности формы.

Свойство Canvas доступно при выполнении программы, поэтому получаемые с его помощью рисунки являются *динамическими* и существуют только в процессе работы приложения. При необходимости можно сохранить рисунок в графическом файле или вывести на принтер. Создаваемые при выполнении программы рисунки могут быть неподвижными или анимационными, т. е. изменяющими свои размеры, форму и расположение.

При выполнении различных *графических операций* используются типы TPoint и TRect, описанные так:

```
TPoint = record
  X: Longint;
  Y: Longint;
  end;
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
  end;
```

Тип TPoint используется для задания *координат точки*, а тип TRect служит для определения *прямоугольной области* путем указания координат левого верхнего и правого нижнего углов.

Поверхность рисования (класс *TCanvas*)

Поверхность рисования представляет собой объект класса TCanvas, иногда его называют *холстом*. У холста есть много свойств и методов, позволяющих перемещаться по поверхности рисования, отображать графические примитивы, копировать изображения и их отдельные области, а также выводить текстовую информацию.

Любая поверхность рисования включает в себя объекты "перо" TPen, "кисть" TBrush и "шрифт" TFont. Объекты "перо" и "кисть" используются для *прорисовки* и *заполнения* геометрических фигур, а объект "шрифт" позволяет управлять *атрибутами текста*, выводимого на поверхности.

Отметим, что компонент, имеющий свойство Canvas, в свою очередь сам может содержать объекты "перо", "кисть" и "шрифт" и, соответственно, иметь свойства Pen, Brush и Font. Таким образом, например, свойство шрифта формы Form1.Font не совпадает со свойством шрифта поверхности рисования формы Form1.Canvas.Font. Шрифт формы задает размер символов для элементов управления формы, если у них установлено значение True свойства ParentFont. Шрифт формы можно устанавливать при проектировании приложения или динамически — при его выполнении. Шрифт поверхности рисования формы определяет размер символов текста, отображаемого в форме с помощью класса TCanvas. Шрифт поверхности рисования формы, как и инкапсулирующий его объект TCanvas, доступен только при выполнении программы.

При выполнении графических операций используется *текущий указатель* (указатель позиции). Он представляет собой невидимый маркер, определяющий позицию на поверхности рисования, начиная с которой выполняется следующая графическая операция. Текущая позиция определяется горизонтальной (`X`) и вертикальной (`Y`) координатами. По умолчанию начало системы координат находится в левом верхнем углу поверхности рисования, а отсчет координат осуществляется в пикселах.

Для *перемещения текущего указателя* в новую позицию можно использовать метод `MoveTo(X, Y: Integer)`. В результате выполнения этой процедуры перо устанавливается в новую позицию холста с координатами `X` и `Y`. При таком перемещении на холсте ничего не рисуется. Положение текущего указателя также изменяют методы, связанные с выводом на холст фигур и текста: при их выполнении текущий указатель остается в позиции, где завершается процесс вывода.

Для рисования геометрических фигур используются следующие методы:

- `Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)` (дуга);
- `Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)` (линия, соединяющая две точки эллипса, хорда);
- `Ellipse(X1, Y1, X2, Y2: Integer)` (эллипс с заполнением);
- `FillRect(const Rect: TRect)` (прямоугольник с заполнением);
- `FrameRect(const Rect: TRect)` (незаполненный прямоугольник, рамка);
- `LineTo(X, Y: Integer)` (линия от указателя до точки с координатами `X` и `Y`);
- `Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)` ("пирог");
- `Polygon(const Points: array of TPoint)` (многоугольник с заполнением);
- `PolyLine(const Points: array of TPoint)` (незаполненный многоугольник);
- `Rectangle(X1, Y1, X2, Y2: Integer)` (заполненный прямоугольник);
- `RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer)` (заполненный прямоугольник со скругленными краями).

У методов `Arc`, `Chord` и `Ellipse` параметры `X1` и `Y1` задают координаты левого верхнего угла, а параметры `X2` и `Y2` — координаты правого нижнего угла прямоугольника, ограничивающего дугу, хорду или эллипс соответственно. Параметры `X3` и `Y3`, `X4` и `Y4` определяют координаты начальной и конечной точек дуги или хорды.

У методов `Rectangle` и `RoundRect` параметры `X1` и `Y1` задают координаты левого верхнего угла, а параметры `X2` и `Y2` — координаты правого нижнего угла прорисовываемого прямоугольника.

У методов `Polygon` и `PolyLine` параметр `Points` представляет собой массив с координатами вершин многоугольника.

У метода `Pie` параметры `X1`, `Y1`, `X2`, `Y2` задают координаты ограничивающего фигуру прямоугольника, а параметры `X3` и `Y3`, `X4` и `Y4` определяют координаты первой и второй линий радиуса соответственно.

Параметры линий фигур и их заполнение определяют текущие значения свойств пера и кисти поверхности рисования.

Рассмотрим пример рисования на поверхности формы с использованием свойства `Canvas`. Выполним вывод на поверхность формы изображение пейзажа с домиком (рис. 10.2), который создадим из эллипсов, прямоугольников и многоугольников. Цвет и заливка (заполнение) будут устанавливаться с помощью свойств объектов "кисть" и "перо".

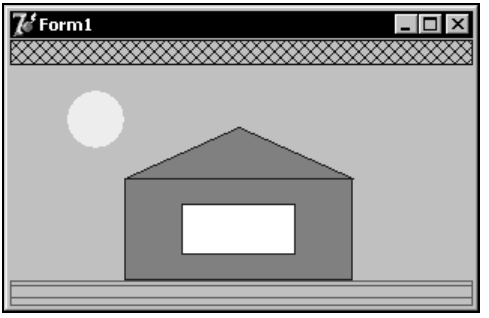


Рис. 10.2. Рисунок на поверхности формы

В листинге 10.1 приведен исходный код модуля, реализующего вывод изображения.

Листинг 10.1. Пример рисования на поверхности формы

```
unit uHome;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    procedure FormResize(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormResize(Sender: TObject);
var w, h, wm, hm: integer;
begin
  Form1.Refresh;
  wm := Form1.ClientWidth;
  w := wm div 8;
  hm := Form1.ClientHeight;
  h := hm div 10;
  with Form1.Canvas do begin
     // Небо
     Brush.Color := clBlue;
```

```
      Brush.Style := bsDiagCross;
      Pen.Color := clBlue;
      Rectangle(0, 0, wm, h);
      // Трава
      Brush.Color := clGreen;
      Brush.Style := bsHorizontal;
      Pen.Color := clGreen;
      Rectangle(0, hm - h, wm, hm);
      // Солнце
      Brush.Color := clYellow;
      Brush.Style := bsSolid;
      Pen.Color := clYellow;
      Ellipse(w, 2 * h, 2 * w, 2 * h + w);
      // Домик
      Brush.Color := clGray;
      Brush.Style := bsSolid;
      Pen.Color := clMaroon;
      Rectangle(2 * w, hm - 5 * h, 6 * w, hm - h);
      Polygon([Point(2 * w, hm - 5 * h), Point(4 * w, hm - 7 * h),
               Point(6 * w, hm - 5 * h), Point(2 * w, hm - 5 * h)]);
      Brush.Color := clWhite;
      Brush.Style := bsSolid;
      Pen.Color := clMaroon;
      Rectangle(3 * w, hm - 4 * h, 5 * w, hm - 2 * h);
  end;
end;
end.
```

Код, выполняющий отображение рисунка, расположен в обработчике события `OnResize` формы, поэтому при изменении ее размеров рисунок перерисовывается заново, в том числе после ее первоначального создания. Методы, выполняющие построение изображения, используют относительные (т. е. заданные относительно высоты и ширины формы) координаты точек, поэтому рисунок всегда занимает всю клиентскую область формы независимо от ее размеров.

Помимо рисования отдельных примитивов, с помощью методов `CopyRect` и `Draw` можно также отобразить заранее *подготовленное изображение*, находящееся на другой поверхности рисования или в другом графическом объекте.

Процедура `CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect)` *копирует прямоугольную область*, размеры которой заданы параметром `Source`, из исходной поверхности, заданной параметром `Canvas`, в прямоугольную область `Dest` результирующей поверхности рисования. Адресатом операции копирования является поверхность, для которой была вызвана процедура. Если исходная и результирующая поверхности имеют разные размеры, то копируемый образ масштабируется под размеры области `Dest`, т. е. результирующей поверхности.

Например, в процедуре

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Form1.Canvas.CopyRect(Form1.Canvas.ClipRect,
                        Image1.Canvas, Image1.Canvas.ClipRect);
end;
```

изображение из компонента Image1 копируется в форму Form1, занимая всю ее поверхность рисования. Если изменить размер формы и нажать кнопку Button1, то изображение опять займет всю поверхность клиентской области формы.

Свойство ClipRect типа TRect определяет *границы области* поверхности рисования *(границы отсечения)*, в пределах которой выполняется прорисовка. Часть изображения, выходящая за пределы этого прямоугольника, отсекается и не выводится. Это свойство можно использовать для ограничения прорисовываемой области, чтобы ускорить процесс отображения. По умолчанию границы области отсечения устанавливаются по размерам поверхности компонента. Однако для компонента Image первоначальные размеры области отсечения равны размерам загруженного в него изображения, а для формы — размерам клиентской области.

Вывод изображения на поверхность рисования осуществляет процедура Draw(X, Y: Integer; Graphic: TGraphic). В ней параметры X и Y определяют левый верхний угол начала вывода, а параметр Graphic задает выводимое изображение типа "растровый массив", "метафайл" или "значок". Занимаемая изображением область зависит от его размеров, и возможна ситуация, когда рисунок не помещается на холсте или закрывает другие объекты. В этом случае лучше использовать процедуру StretchDraw(const Rect: TRect; Graphic: TGraphic), в которой для изображения задается не начальная точка, а прямоугольная область вывода Rect.

В процедуре:

```
procedure TForm1.Button2Click(Sender: TObject);
var picture: TBitmap;
begin
  picture := TBitmap.Create;
  try
    picture.LoadFromFile('photo.bmp');
    Form1.Canvas.Draw(10, 20, picture);
  finally
    Ris.Free;
  end;
end;
```

изображение читается из файла photo.bmp и отображается на поверхности формы.

Следующие методы и свойства предназначены для операций вывода на поверхность рисования текстовой информации.

Метод TextOut(X, Y: Integer; const Text: String) служит для *отображения текста* на поверхности рисования. Эта процедура выводит строку текста, заданную параметром Text, при этом координаты X и Y устанавливают левый верхний угол области вывода. Параметры шрифта определяются текущим значением свойства Font поверхности, на которой отображается текст. При вызове процедуры TextOut программист должен указать, что используется метод объекта TCanvas. В противном случае будет вызвана одноименная API-функция TextOut, также предназначенная для вывода текстовой информации, но отличающаяся количеством и типом параметров. Из-за этих различий при компиляции выдается сообщение об ошибке.

Пример вывода текста на поверхности формы:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  // Вывод текста в форме
  Canvas.TextOut(100, 100, 'Тект');
  // Другой вариант инструкции вывода текста в форме:
  // Form1.Canvas.TextOut(100, 100, 'Тект');
  // Этот вызов приведет к ошибке компиляции:
  // TextOut(100, 100, 'Тект');
end;
```

Вывод текста на холст можно выполнить также с помощью метода `TextRect`, отличающегося от метода `TextOut` наличием дополнительного параметра `Rect`, который ограничивает область вывода.

В ряде случаев для *управления размещением текста* на поверхности рисования требуется определить размеры прямоугольника, занимаемого выводимой строкой. Для этого удобно использовать методы `TextHeight`, `TextWidth` и `TextExtent`.

Функции `TextHeight(const Text: String): Integer` и `TextWidth(const Text: String): Integer` возвращают в качестве результирующих значений (в пикселах) соответственно *высоту* и *ширину* прямоугольной области, занимаемой строкой `Text`.

Функция `TextExtent(const Text: String): TSize` возвращает запись, содержащую значение (в пикселах) высоты `cy` и ширины `cx` области вывода. Тип записи `TSize` описан так:

```
type  TSize = record
        cx: Longint;
        cy: Longint;
      end;
```

Размеры области вывода текста зависят от шрифта и его параметров. Например, при увеличении размера шрифта (свойство `Canvas.Font.Size`) соответственно увеличиваются высота и ширина области вывода.

Важным свойством объекта класса `TCanvas` является свойство `Pixels[X,Y: Integer]` типа `TColor`, которое определяет цвет пиксела в точке, указанной индексами `X` и `Y` двумерного массива координат `Pixels`. Это свойство доступно как для чтения, так и для записи.

Пример использования свойства `Pixels`:

```
procedure TForm1.Button2Click(Sender: TObject);
var n, x, y :integer;
begin
  Randomize;
  for n := 1 to 20 do begin
    x := Random(Form1.ClientWidth);
    y := Random(Form1.ClientHeght);
    Form1.Canvas.Pixels[x,y] := clWhite;
  end;
end;
```

Приведенная процедура выполняет вывод на поверхность формы двадцати случайно расположенных белых точек.

Объекты класса `TCanvas` включают в себя объекты `TFont`, `TPen` и `TBrush` и, соответственно, имеют свойства `Font`, `Pen` и `Brush`. Отметим, что данные объекты могут также входить в состав других визуальных компонентов, например, формы или панели.

Свойство `Font` типа `TFont` устанавливает *параметры шрифта*, применяемого для отображения текста на поверхности рисования. Управление параметрами шрифта осуществляется через его свойства, основными из которых являются следующие:

- `Name` типа `TFontName` (название шрифта, например, Arial или Times New Roman). Отметим, что свойство `Name` шрифта не связано с одноименным свойством самого компонента;
- `Size` типа `Integer` (размер шрифта в пунктах). Пункт равен 1/72 дюйма;
- `Height` типа `Integer` (размер шрифта в пикселах). Если значение этого свойства является положительным числом, то в него включен и межстрочный интервал. Если размер шрифта имеет отрицательное значение, то интервал не учитывается;
- `Style` типа `TFontStyle` (стиль шрифта), может принимать комбинации следующих значений:
  - `fsItalic` (курсив);
  - `fsBold` (полужирный);
  - `fsUnderline` (подчеркнутый);
  - `fsStrikeOut` (перечеркнутый);
- `Color` типа `TColor` (цвет выводимого текста).

Свойства `Size` и `Height` являются взаимозависимыми: при установке значения одного из них второе свойство автоматически получает соответствующее значение.

Так, в примере

```
// Установка параметров шрифта
Form1.Canvas.Font.Name := 'Courier';
Form1.Canvas.Font.Style := [fsBold] + [fsUnderline];
Form1.Canvas.Font.Size := 14;
Form1.Canvas.Font.Color := clRed;
// Вывод текста шрифтом с установленными параметрами
Form1.Canvas.TextOut(100, 100, 'Управление параметрами шрифта');
```

на поверхность формы выводится полужирный подчеркнутый текст шрифтом Courier размером 14 пунктов с красным цветом символов.

Для получения *списка доступных* (установленных в системе) *шрифтов* можно использовать свойство `Fonts` глобального объекта `Screen` экрана.

Для иллюстрации рассмотрим следующий пример. В списке `ListBox1` содержится перечень доступных шрифтов (рис. 10.3). При выборе какого-либо шрифта он устанавливается в качестве шрифта для формы `Form1`, что отражается в ее заголовке. При этом шрифт компонентов `Label1`, `Edit1` и `Button1` изменяется на выбранный. Чтобы измене-

ние шрифта формы автоматически приводило к смене шрифта указанных компонентов, их свойство `ParentFont` установлено в значение `True`.

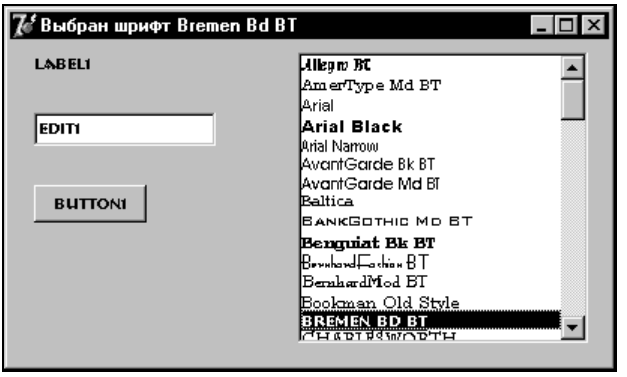


Рис. 10.3. Работа со списком шрифтов

В листинге 10.2 приводится текст модуля `uFonts` формы `Form1`.

Листинг 10.2. Пример работы со шрифтами

```
unit uFonts;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
       Edit1: TEdit;
     Button1: TButton;
      Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure ListBox1DrawItem(Control: TWinControl; Index: Integer;
              Rect: TRect; State: TOwnerDrawState);
    procedure ListBox1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  // Загрузка списка доступных шрифтов
  ListBox1.Items := Screen.Fonts;
```

```
  // Первоначально в списке ни один элемент не выбран
  ListBox1.ItemIndex := -1;
  ListBox1.Style := lbOwnerDrawFixed;
  Label1.ParentFont := True;
  Edit1.ParentFont := True;
  Button1.ParentFont := True;
end;

procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
          Rect: TRect; State: TOwnerDrawState);
begin
  TListBox(Control).Canvas.FillRect(Rect);
  TListBox(Control).Canvas.Font.Name := TListBox(Control).Items[Index];
  TListBox(Control).Canvas.TextOut(Rect.Left,Rect.Top,
                       TListBox(Control).Items[Index]);
end;

procedure TForm1.ListBox1Click(Sender: TObject);
begin
  Form1.Caption := 'Выбран шрифт ' + ListBox1.Items[ListBox1.ItemIndex];
  // Смена шрифта для тех компонентов, расположенных в форме,
  // у которых свойство ParentFont установлено в значение True
  Form1.Font.Name := ListBox1.Items[ListBox1.ItemIndex];
end;
```

Прорисовка области списка `ListBox1` осуществляется программно, для этого свойство `Style` списка устанавливается в значение `lbOwnerDrawFixed`. Код операций вывода строк списка размещен в обработчике события `OnDrawItem`.

После запуска программы ни один из элементов списка (шрифтов) не выбран, и для вывода текста формы и ее интерфейсных компонентов используется шрифт, установленный при разработке программы. Поскольку прорисовка списка `ListBox1` выполняется программно, то смена шрифта формы на этот компонент не влияет.

Для удобства выбора и установки параметров шрифта можно использовать стандартное диалоговое окно `FontDialog`.

Свойство `Pen` типа `TPen` определяет *атрибуты пера*, применяемого для рисования линий и границ геометрических фигур. Управление атрибутами пера осуществляется через его свойства, основными из которых являются следующие:

- `Color` типа `TColor` (цвет пера);
- `Style` типа `TPenStyle` (стиль рисуемой линии); может принимать следующие значения:
  - `psSolid` (сплошная линия) — по умолчанию;
  - `psDash` (штриховая линия);
  - `psDot` (пунктирная линия);
  - `psDashDot` (штрихпунктирная линия);

- `psDashDotDot` (линия вида штрих-точка-точка);
- `psClear` (невидимая линия);
- `psInsideFrame` (линия внутри прямоугольника поверхности рисования);

♦ `Width` типа `Integer` (толщина рисуемой линии в пикселах);

♦ `Mode` типа `TPenMode` (способ, в соответствии с которым перо при рисовании линии взаимодействует с имеющимися на холсте пикселами). Свойство `Mode` принимает следующие значения, определяющие цвет вновь рисуемой линии:

- `pmBlack` (всегда черный);
- `pmWhite` (всегда белый);
- `pmNop` (не изменяется);
- `pmNot` (инверсия цвета на поверхности рисования);
- `pmCopy` (совпадает с цветом, указанным в свойстве `Color`) — по умолчанию;
- `pmNotCopy` (инверсия цвета пера);
- `pmMergePenNot` (объединение цвета пера и инверсии цвета поверхности рисования);
- `pmMaskPenNot` (объединение общих цветов инверсии поверхности рисования и пера);
- `pmMergeNotPen` (объединение цвета поверхности рисования и инверсии цвета пера);
- `pmMaskNotPen` (объединение общих цветов поверхности рисования и инверсии пера);
- `pmMerge` (объединение цвета поверхности рисования и пера);
- `pmNotMerge` (инверсия цвета, получаемого при значении `pmMerge`);
- `pmMask` (объединение общих цветов поверхности рисования и пера;
- `pmNotMask` (инверсия цвета, получаемого при значении `pmMask`);
- `pmXor` (объединение цветов, различных для поверхности рисования и пера);
- `pmNotXor` (инверсия цвета, получаемого при значении `pmXor`).

Так, в примере:

```
// Установка атрибутов пера
Form1.Canvas.Pen.Mode := pmCopy;
Form1.Canvas.Pen.Style := psDot;
Form1.Canvas.Pen.Width := 2;
Form1.Canvas.Pen.Color := clGreen;
// Рисование линии пером с установленными атрибутами
Form1.Canvas.LineTo(50, 100);
```

на поверхности формы пером рисуется зеленая пунктирная линии толщиной 2 пиксела.

Свойство `Brush` типа `TBrush` определяет *вид заполнения* геометрических фигур, например, прямоугольника или эллипса. Управление заполнением фигур осуществляется через свойства кисти, основными из которых являются следующие:

- `Color` типа `TColor` (цвет кисти);
- `Style` типа `TBrushStyle` (стиль кисти); может принимать следующие значения:
  - `bsSolid` (сплошная заливка);
  - `bsClear` (нет заливки);
  - `bsHorizontal` (параллельные горизонтальные линии);
  - `bsVertical` (параллельные вертикальные линии);
  - `bsFDiagonal` (параллельные диагональные линии, направленные вверх);
  - `bsBDiagonal` (параллельные диагональные линии, направленные вниз);
  - `bsCross` (прямая решетка);
  - `bsDiagCross` (косая решетка);
- `Bitmap` типа `TBitmap` (растровое изображение, используемое в качестве стиля кисти).

В этом примере

```
// Установка параметров кисти
Form1.Canvas.Brush.Style := bsHorizontal;
Form1.Canvas.Brush.Color := clBlue;
// Вывод эллипса, заполненного узором
Form1.Canvas.Ellipse(10, 20, 50, 100);
```

рисование осуществляется кистью: на поверхность формы выводится эллипс, заполненный параллельными горизонтальными линиями синего цвета.

Кроме стандартных узоров, задаваемых свойством `Style`, для заполнения фигур можно использовать растровое изображение из файла формата BMP. Например, в процедуре:

```
procedure TForm1.Button1Click(Sender: TObject);
var Pattern: TBitmap;
begin
  // Чтение изображения из файла
  Pattern := TBitmap.Create;
  try
    Pattern.LoadFromFile('Pattern.bmp');
    Form1.Canvas.Brush.Bitmap := Pattern;
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Pattern.Free;
  end;
  // Вывод круга, заполненного заданным изображением
  Form1.Canvas.Ellipse(50, 50, 100, 100);
end;
```

в качестве заполнения фигур используется растровое изображение из файла Pattern.bmp. Изображение можно подготовить с помощью любого графического редактора, в том числе Image Editor версии 3.0, поставляемого в комплекте с Delphi.

Для определения *текущего положения пера* используется свойство `PenPos` типа `TPoint`. Например, инструкция

```
Form1.Canvas.MoveTo(Form1.Canvas.PenPos.X + 30,Form1.Canvas.PenPos.Y — 20);
```

перемещает перо на 30 пикселов вправо и на 20 пикселов вверх.

Перемещение пера можно выполнить и так:

```
Form1.Canvas.PenPos.X := Form1.Canvas.PenPos.X + 30;
Form1.Canvas.PenPos.Y := Form1.Canvas.PenPos.Y — 20;
```

При *изменении содержимого поверхности* рисования генерируются события `OnChanging` и `OnChange` типа `TNotifyEvent`. Первое из них возникает *перед* изменением поверхности, а второе — *после*. Отметим, что эти события генерируются только при использовании методов, связанных с рисованием и выводом текста, перемещение указателя с помощью метода `MoveTo` событиями `OnChanging` и `OnChange` не сопровождается.

Так как свойство `Canvas` на этапе проектирования недоступно, обработчики его событий программируются вручную. В листинге 10.3 приводится пример такого обработчика (события изменения поверхности рисования формы).

Листинг 10.3. Пример обработчика события изменения поверхности рисования

```
unit uCanvas;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    Button2: TButton;
    procedure FormCanvasChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCanvasChange(Sender: TObject);
begin
  Form1.Caption := TimeToStr(Time);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Canvas.OnChange := FormCanvasChange;
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Form1.Canvas.Pixels[10,20] := clRed;
end;
end.
```

При изменении поверхности рисования формы `Form1` в заголовке формы отображается время последнего изменения. В качестве операции рисования используется вывод на поверхность формы красной точки, выполняемый в обработчике события нажатия кнопки `Button2`. Событие `OnChange` генерируется при каждом выводе точки, несмотря на то, что цвет точки в форме изменяется только при первом нажатии кнопки `Button2`.

Свойством `Canvas` обладают все визуальные компоненты, но не для всех визуальных компонентов это свойство программно доступно (описано со спецификатором доступа `Public`). В случае, когда требуется рисование на поверхности некоторого компонента, а его свойство `Canvas` недоступно, можно применить один из следующих приемов:

- разместить в данном компоненте другой элемент, который предоставляет свойство `Canvas`, например, `Image` или `PaintBox`, и рисовать на этом элементе;
- использовать специальную переменную типа `TCanvas`, которую следует через ее свойство `Handle` связать с контекстом отображения данного компонента, и выполнять рисование на поверхности связанного с этой переменной объекта.

Рассмотрим пример, в котором отображение графики происходит на рабочем столе Windows.

```
procedure TForm1.Button2Click(Sender: TObject);
var dtC: TCanvas;
begin
  dtC :=  TCanvas.Create;
  // Установление связи между dtC и рабочим столом.
  // Константа Hwnd_Desktop содержит ссылку на рабочий стол Windows
  dtC.Handle := GetDC(Hwnd_Desktop);
  // Операции рисования на рабочем столе
  dtC.TextOut(200, 400, 'Вывод графики на рабочем столе');
  // Разрыв связи между dtC и рабочим столом
  ReleaseDC(dtC.Handle, Hwnd_Desktop);
  dtC.Free;
end;
```

Для рисования на поверхности рабочего стола объявляется специальная переменная `dtC` типа `TCanvas`. После создания экземпляра этой переменной устанавливается связь с рабочим столом через его контекст отображения. После этого через переменную `dtC` можно собственно рисовать на поверхности связанного с ней объекта. В качестве операции отображения графики взят вывод строки текста `Вывод графики на рабочем столе`. На практике эти операции могут быть более сложными и использовать все возможности класса `TCanvas`. Допускается также создание анимационных изображений. После завершения операций отображения связь между переменной и рабочим столом разрывается, а экземпляр объекта `dtC` удаляется из памяти.

При выполнении различных графических операций часто используются тип `TColor` и соответствующее ему свойство `Color`, предназначенные для *управления цветом* объек-

тов. Тип `TColor` уже был рассмотрен ранее в *главе 3*, посвященной основным визуальным компонентам и их характеристикам. Для управления цветом со стороны пользователя удобно применять стандартное диалоговое окно `ColorDialog`.

Для работы со *значениями цветов* программист может использовать специальные GDI-функции (например, `ColorToRGB`, `GetRValue`, `GetGValue`, `GetBValue` и `RGB`), предназначенные для получения и преобразования цвета и его составляющих (красной, зеленой и синей).

Кроме цветовых, в распоряжении программиста имеется множество других GDI-функций, связанных с графическими операциями. В параметрах этих функций используется свойство `Handle` объекта `TCanvas`, которое представляет собой дескриптор контекста устройства, в данном случае поверхности рисования.

Анимация

Под *анимацией* понимается перемещение и изменение формы различных изображений на экране. В основе перемещения какого-либо объекта на экране лежит следующая последовательность действий:

1. Вывести объект на экран.
2. Стереть объект с экрана.
3. Вывести с некоторым смещением другой вариант объекта и т. д.

При частом выводе объекта с небольшими смещениями создается иллюзия его движения.

Есть много анимационных алгоритмов, отличающихся способом вывода объектов, источниками поступления данных об изображении и т. п. Простейший анимационный алгоритм включает в себя следующие шаги:

1. Выводится рисунок определенным цветом.
2. Рисунок формируется на том же месте цветом, совпадающим с цветом фона, т. е. как бы "исчезает".
3. Рисунок выводится на другом месте первоначальным цветом и т. д.

Пример рисования движущегося мяча:

```
procedure TForm1.Button1Click(Sender: TObject);
label 10;
const br = 30;
      bc = clRed;
begin
  bx :=  Form1.ClientWidth div 2;
  by :=  Form1.ClientHeight - br;
10:
  if bx > Form1.ClientWidth - br then exit;
  with Canvas do begin
     Pen.Color := Form1.Color;
     Brush.Color :=  Form1.Color;
     Ellipse(bx - br,by - br,bx + br,by + br);
  end;
```

```
  bx := bx + 1;
  with Canvas do begin
     Pen.Color := bc;
     Brush.Color := bc;
     Ellipse(bx — br,by — br,bx + br,by + br);
  end;
  goto 10;
end;
```

Мяч, представленный окружностью красного цвета, перемещается по горизонтали от середины до правой границы формы. Процедура является обработчиком события нажатия кнопки `Button1`.

Анимационные алгоритмы можно применить и к визуальным элементам, в том числе к рассматриваемым далее графическим компонентам. При этом алгоритм упрощается, т. к. для перемещения компонента достаточно только указать координаты (значения свойств `Top` и `Left`) его нового положения, стирать предыдущее изображение не требуется.

В качестве примера рассмотрим приложение, реализующее стрелочные часы.

В левой части формы (рис. 10.4) расположен циферблат со стрелками, в правой — кнопка `Button1`, предназначенная для запуска и остановки процесса отсчета времени.

Размер часов рассчитывается исходя из вертикального размера формы. Отображение циферблата и стрелок выполняется с помощью свойства `Canvas`, для рисования используется поверхность формы `Form1`.

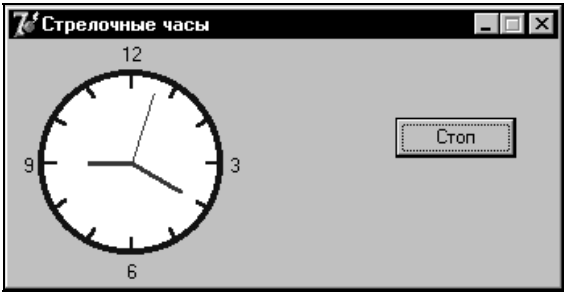


Рис. 10.4. Пример анимации: часы со стрелками

В листинге 10.4 приводится код модуля `uArClock` главной формы приложения.

Листинг 10.4. Пример анимации: часы со стрелками

```
unit uArClock;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
     Timer1: TTimer;
```

```
    procedure DrawArrows(DrawColor: TColor);
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormPaint(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var                          Form1: TForm1;
          CenterX, CenterY, Radius: integer;
     HourArrow, MinArrow, SecArrow: integer;
               Hour, Min, Sec, MSec: word;
     HourAngle, MinAngle, SecAngle: real;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.BorderStyle := bsSingle;
  Form1.BorderIcons := [biSystemMenu, biMinimize];
  Timer1.Interval := 1000;
  Timer1.Enabled := False;
  CenterY := Form1.ClientHeight div 2;
  CenterX := CenterY;
  Radius := CenterX - 20;
  HourArrow := Radius - 30;
  MinArrow := Radius - 20;
  SecArrow := Radius - 10;
end;

procedure TForm1.FormPaint(Sender: TObject);
var i: integer;
begin
  // Прорисовка циферблата
  with Form1.Canvas do begin
     // Вывод окружности
     Pen.Color := clBlue;
     Pen.Width := 4;
     Brush.Color := clWhite;
     Ellipse(20, 20, 20 + 2 * Radius, 20 + 2 * Radius);
     Pen.Width := 2;
     // Вывод рисок
     for i := 0 to 11 do begin
         MoveTo(CenterX + Round((Radius - 9) * sin(i / 6 * pi)),
                CenterY - Round((Radius - 9) * cos(i / 6 * pi)));
         LineTo(CenterX + Round((Radius) * sin(i / 6 * pi)),
                CenterY - Round((Radius) * cos(i / 6 * pi)));
     end;
```

```
    // Вывод цифр
    Font.Height := 10;
    Font.Color  := clBlack;
    Brush.Color := Form1.Color;
    TextOut(CenterX - TextWidth('12') div 2, CenterY - Radius -
            TextHeight('12') - 5,'12');
    TextOut(CenterX + Radius + 5, CenterY - TextHeight('3'), '3');
    TextOut(CenterX - TextWidth('6') div 2, CenterY + Radius + 5, '6');
    TextOut(CenterX - Radius - TextWidth('9') - 5,
            CenterY - TextHeight('9'), '9');
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Button1.Caption = 'Старт' then begin
    // Запуск часов
    DecodeTime(Time, Hour, Min, Sec, MSec);
    HourAngle := (Hour mod 12) / 12 * (2 * Pi);
    MinAngle := Min / 60 * (2 * Pi);
    SecAngle := Sec / 60 * (2 * Pi);
    DrawArrows(clRed);
    Button1.Caption := 'Стоп';
    Timer1.Enabled := True;
  end
  else begin
    // Остановка часов
    Button1.Caption := 'Старт';
    Timer1.Enabled  := False;
  end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  // Стереть стрелки
  DrawArrows(clWhite);
  // Нарисовать стрелки на новом месте
  DecodeTime(Time, Hour, Min, Sec, MSec);
  HourAngle := (Hour mod 12) / 12 * (2 * Pi);
  MinAngle := Min / 60 * (2 * Pi);
  SecAngle := Sec / 60 * (2 * Pi);
  DrawArrows(clRed);
end;

procedure TForm1.DrawArrows(DrawColor: TColor);
begin
  with Form1.Canvas do begin
     Pen.Color := DrawColor;
     MoveTo(CenterX, CenterY);
     // Часовая стрелка
     Pen.Width := 3;
```

```
      LineTo(CenterX + Round(HourArrow * sin(HourAngle)),
             CenterY – Round(HourArrow * cos(HourAngle)));
      MoveTo(CenterX, CenterY);
      // Минутная стрелка
      Pen.Width := 2;
      LineTo(CenterX + Round(MinArrow * sin(MinAngle)),
             CenterY – Round(MinArrow * cos(MinAngle)));
      MoveTo(CenterX, CenterY);
      // Секундная стрелка
      Pen.Width := 1;
      LineTo(CenterX + Round(SecArrow * sin(SecAngle)),
             CenterY – Round(SecArrow * cos(SecAngle)));
  end;
end;
end.
```

Код, выполняющий прорисовку циферблата, расположен в обработчике события `OnPaint` формы, чтобы изображение часов восстанавливалось автоматически. Событие `OnPaint` генерируется каждый раз, когда форма требует перерисовки, например, если окно выходит на первый план после того, как оно (или его часть) было закрыто другими окнами. При обработке события `OnPaint` автоматически вызываются методы `Invalidate` и `Update`. Если необходима принудительная прорисовка поверхности формы, то программист может вызывать эти методы явно. Однако их вызовы нельзя располагать в обработчике события `OnPaint`, т. к. в этом случае будет происходить непрерывный рекурсивный вызов данных методов, что является ошибкой.

Выполнять рисование с помощью методов класса `TCanvas` в обработчике события `OnCreate` бесполезно, т. к. на момент возникновения этого события форма еще не создана.

Для отсчета времени используется таймер `Timer1`, для которого установлен интервал, равный `1000`, что соответствует генерации одного события `OnTimer` в секунду. В обработчике события `OnTimer` с помощью процедуры `DecodeTime` выполняется преобразование текущего значения времени в формат целых чисел (тип `word` тоже относится к целочисленному типу), соответствующих значениям часов, минут и секунд. На основе значений составляющих времени определяются значения углов поворота для часовой, минутной и секундной стрелок, затем с помощью процедуры `ArrowDraw` организуется прорисовка стрелок. Весь код, относящийся к процедуре `DrawArrows`, подготавливается программистом самостоятельно.

Пользователь не имеет возможности изменять размер формы, т. к. ее свойство `BorderStyle` установлено в значение `bsSingle`. Если разрешить пользователю управлять размером формы, то программист должен предусмотреть для этой формы обработку события `OnResize`, соответственно изменяя размеры часов. В случае изменения размеров формы для удаления старого изображения перед перерисовкой циферблата следует очистить поверхность формы, например, с помощью метода `Refresh`.

Если объединить в одной форме стрелочные часы и цифровые, программирование которых было рассмотрено ранее, то можно создать приложение-часы, напоминающее программу clock.exe из состава Windows.

Для создания анимационных эффектов можно также использовать специальные компоненты `Animate` и `MediaPlayer`, рассматриваемые в *главе 11*, посвященной мультимедийным возможностям Delphi.

Графические компоненты

При конструировании формы для создания визуальных эффектов и изображений можно использовать соответствующие компоненты. Действия, связанные с построением изображения, напоминают работу в среде графического редактора.

Подобным образом можно создавать относительно простые визуальные эффекты, такие как отображение рамок, фасок или элементарных геометрических фигур. Это связано с тем, что набор компонентов, из которых конструируется изображение, невелик, а возможности этих компонентов ограниченны. Наиболее часто используются такие графические компоненты, как геометрическая фигура (`Shape`), фаска (`Bevel`), графическое изображение (`Image`).

Графические элементы являются *неоконными визуальными компонентами* и происходят от класса `TGraphicControl`.

Компонент *Shape*

Для отображения геометрических фигур в Delphi служит компонент `Shape`. *Вид фигуры* (рис. 10.5), отображаемой этим компонентом, определяется одноименным свойством `Shape` типа `TShapeType`, принимающим следующие значения:

- `stCircle` (круг);
- `stEllipse` (эллипс);
- `stRectangle` (прямоугольник);
- `stRoundRect` (прямоугольник со скругленными углами);
- `stRoundSquare` (квадрат со скругленными углами);
- `stSquare` (квадрат).

Управление *цветом* и *заполнением* фигуры выполняется с помощью свойств `Pen` и `Brush`.

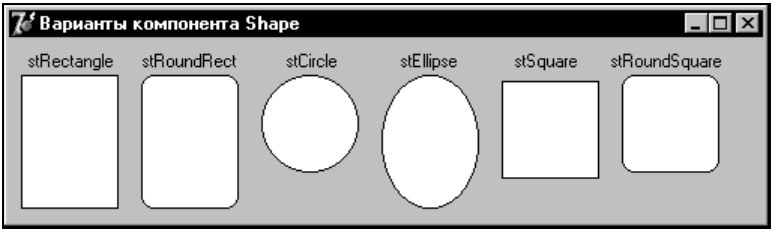


Рис. 10.5. Фигуры, отображаемые компонентом `Shape`

Компонент *Bevel*

Фаски представляют собой прямоугольные области, рамки и линии. Они имеют плоский или объемный вид и обычно используются для визуального выделения других элементов формы с целью более наглядного их восприятия. Для работы с фаской в Delphi служит компонент `Bevel`.

Фигура (рис. 10.6), используемая для фаски, задается свойством `Shape` типа `TBevelShape`, принимающим следующие значения:

- `bsBox` (прямоугольник);
- `bsFrame` (рамка);
- `bsTopLine` (линия сверху);
- `bsBottomLine` (линия снизу);
- `bsLeftLine` (линия слева);
- `bsRightLine` (линия справа);
- `bsSpacer` (прямоугольная область, невидимая при выполнении программы).

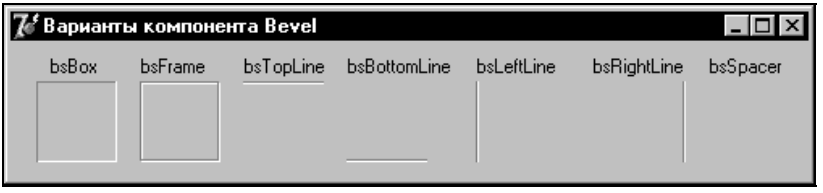


Рис. 10.6. Виды фасок, отображаемых компонентом `Bevel`

Свойство `Style` типа `TBevelStyle` определяет стиль фаски и принимает следующие значения:

- `bsLowered` (фаска выглядит утопленной относительно поверхности размещения) — по умолчанию;
- `bsRaised` (фаска выглядит приподнятой относительно поверхности размещения).

Компонент *Image*

Данный компонент используется для отображения изображения определенного графического формата. Он обычно помещается на поверхность формы и представляет собой невидимый контейнер для размещения реального изображения. В Delphi графическое изображение представлено компонентом `Image`.

Компонент `Image` включает в себя класс `TPicture`, который, в свою очередь, имеет свойства и методы, используемые для работы с готовыми изображениями. Основным свойством этого компонента является свойство `Picture`. С его помощью можно, например, загрузить изображение.

Свойство `Picture` типа `TPicture` *определяет изображение*, размещаемое внутри компонента `Image`.

Объект типа TPicture является контейнером для графических объектов и может содержать растровое изображение форматов BMP, ICO или WMF. Для этого он включает в себя классы TBitmap, TIcon и TMetaFile. *Графическое изображение*, загруженное в объект типа TPicture, определяется свойством Graphic типа TGraphic. Это свойство можно использовать для доступа к изображениям любого из указанных типов, если он не известен. Если тип графики известен, то для операций с ней можно использовать свойства Bitmap типа TBitmap, Icon типа TIcon и MetaFile типа TMetaFile.

Свойства Height и Width типа Integer определяют соответственно *высоту* и *ширину* загруженного в объект типа TPicture изображения. Отметим, что значения этих свойств в общем случае не равны значениям одноименных свойств компонента Image, задающих размеры самого компонента Image: они совпадут только в случае, если изображение, загруженное в компонент Image с помощью свойства Picture, займет его полностью.

В объект типа TPicture изображение может загружаться из следующих источников:

- графический файл;
- компонент, содержащий изображение, например, Image;
- файл ресурса.

Изображение из графического файла можно загружать на этапе проектирования приложения (статически) и при его выполнении (динамически). Следует учитывать, что изображение, подключенное при проектировании приложения, соответственно увеличивает объем исполняемого файла программы. Чтобы избежать этого, рекомендуется загружать большие изображения динамически.

Для *загрузки изображения из файла* в классе TPicture имеется метод LoadFromFile(const FileName: String), параметр FileName которого указывает графический файл-источник изображения. Данная процедура способна работать с файлами форматов BMP, WMF, ICO.

Например, инструкция Image1.Picture.LoadFromFile('c:\picture\photo1.bmp'); загружает изображение из файла c:\picture\photo1.bmp в компонент Image1.

При *загрузке изображения* из содержащего его компонента для класса TPicture необходимо использовать свойство, указывающее тип графики в объекте-источнике.

Например, инструкция Image2.Picture.Bitmap.Assign(Image1.Picture.Bitmap); копирует изображение типа TBitmap из компонента Image1 в компонент Image2.

Загрузка изображения из файла ресурсов для объекта Bitmap выполняется посредством метода LoadFromResourceName(Instance: THandle; const ResName: String). Файл ресурса (res) может быть подготовлен с помощью любого редактора ресурсов. Перед использованием файл ресурса следует подключить к модулю директивой компилятора $R.

Замечание

В Delphi подключение файлов ресурсов, указанных в директиве {$R ...}, выполняется статически, на этапе компиляции и сборки файлов проекта. Поэтому каждый рисунок, загружаемый через файл ресурса, приводит к увеличению размера исполняемого файла программы. Это относится и к другим видам ресурсов, например, к вариантам указателя мыши.

Пример загрузки изображения из файла ресурса:

```
{$R resource1.res}
...
Image1.Picture.BitMap.LoadFromResourceName(Instance, 'picture1');
```

В компонент `Image1` загружается изображение с именем `picture1`, содержащееся в файле ресурса resource1.res. Имя ресурса, в данном случае `picture1`, задается в редакторе ресурсов при создании изображения.

При необходимости *сохранение изображения*, содержащегося в компоненте, можно выполнить с помощью метода `SaveToFile` класса `TPicture`. Процедура `SaveToFile(const FileName: String)` сохраняет изображение, находящееся в контейнере `Picture`, на диске в файле с именем, заданным параметром `FileName`.

У компонента `Image` есть свойство `Canvas`, поэтому на его поверхности разрешается рисовать даже поверх уже находящегося на нем изображения. Так можно, например, оформить рамку вокруг рисунка или добавить к нему поясняющий текст.

Как уже отмечалось, размеры компонента `Image` и содержащегося в нем изображения (загруженного с помощью свойства `Picture`) в общем случае не совпадают. При этом возможна ситуация, когда изображение не помещается в области компонента `Image`. Для просмотра таких изображений можно использовать свойства `AutoSize` или `Stretch` компонента `Image`.

Свойство `AutoSize` типа `Boolean` управляет возможностью *автоматического приведения размеров* компонента `Image` к размерам содержащегося в нем изображения. Если свойство `AutoSize` установлено в значение `True`, то размеры элемента `Image` приводятся к размерам изображения, если в значение `False` (по умолчанию), то нет.

Свойство `Stretch` типа `Boolean` управляет возможностью *автоматического приведения размеров изображения* к размерам компонента `Image`, в котором оно содержится. Если свойство `Stretch` установлено в значение `True`, то размеры изображения приводятся к размерам компонента `Image`, если в значение `False` (по умолчанию), то размеры изображения не изменяются. Для значков это свойство не действует, они загружаются с исходными размерами.

При приведении размеров изображения к размерам компонента `Image` возможно нарушение пропорций изображения по высоте и ширине. Задав значение `True` для свойства `Proportional` типа `Boolean`, можно обеспечить сохранение пропорций изображения.

Если размеры изображения больше, чем размеры компонента `Image`, а свойства `AutoSize` и `Stretch` установлены в значение `False`, то часть изображения отсекается. Для обеспечения просмотра всего изображения, независимо от размеров области компонента, можно применять указанные далее приемы.

- Установить значение `True` для свойства `Stretch`. При выводе изображения значительных размеров происходит сильная потеря качества рисунка из-за масштабирования. Поэтому такой способ используют, если качество отображения не играет существенной роли, например, в области предварительного просмотра открываемых графических файлов (как в компоненте `OpenPictureDialog1`).

- Установить значение `True` для свойства `AutoSize`. При этом потеря качества рисунка не происходит, т. к. размеры компонента `Image` подстраиваются под изображение, а не наоборот. Однако часто размеры компонента `Image` устанавливаются при разработке или имеют определенные пределы, связанные с дизайном формы, и изменять эти размеры нежелательно. В этом случае можно организовать прокрутку изображения. Для этого компонент `Image` помещают в контейнер (например, панель `Panel`), который ограничивает видимую область этого компонента. Перемещение видимой области осуществляется путем изменения значений свойств `Left` и `Top` компонента `Image`.

Свойство `Center` типа `Boolean` определяет, центрируется ли изображение внутри компонента `Image`. Если свойство установлено в значение `True`, то изображение центрируется, если свойство имеет значение `False` (по умолчанию), то изображение выравнивается по левому верхнему углу компонента `Image`.

Рассмотрим в качестве примера приложение, обеспечивающее просмотр содержимого графических файлов с помощью прокрутки (рис. 10.7).

При нажатии кнопки **Открыть** (`Button1`) появляется реализуемое компонентом `OpenPictureDialog1` диалоговое окно выбора имени графического файла для открытия. Выход из программы осуществляется нажатием кнопки **Выход** (`Button2`).

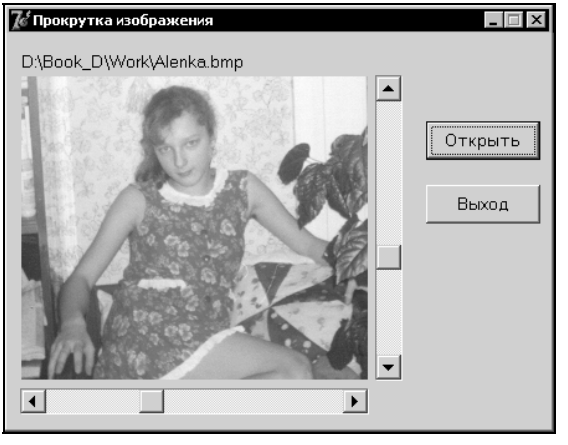


Рис. 10.7. Прокрутка изображения

Компонент `Image1` расположен на панели `Panel1` и занимает всю ее поверхность. Для этого при создании формы свойство `Align` компонента установлено в значение `alClient`. Для обеспечения возможности прокрутки свойству `Align` присвоено значение `alNone`.

В случае успешного выбора графический файл загружается в компонент `Image1`, а имя файла выводится в надписи `Label1`. Кроме того, выполняются определение и установка максимальных значений для горизонтальной (`ScrollBarHor`) и вертикальной (`ScrollBarVert`) полос прокрутки.

В листинге 10.5 приводится код модуля `uImageSc` формы приложения.

Листинг 10.5. Пример прокрутки изображения

```
unit uImageSc;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls, ExtDlgs;
type
  TForm1 = class(TForm)
             Panel1: TPanel;
             Image1: TImage;
       ScrollBarHor: TScrollBar;
      ScrollBarVert: TScrollBar;
            Button1: TButton;
OpenPictureDialog1: TOpenPictureDialog;
            Button2: TButton;
             Label1: TLabel;

    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure ScrollBarHorChange(Sender: TObject);
    procedure ScrollBarVertChange(Sender: TObject);
    procedure Button2Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  BorderIcons := [biSystemMenu,biMinimize];
  Panel1.Caption    := '';
  Image1.AutoSize   := True;
  Image1.Align      := alClient;
  Image1.Align      := alNone;
  ScrollBarHor.Min  := 0;
  ScrollBarVert.Min := 0;
  Label1.Caption    := '';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then begin
    Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
    Label1.Caption := OpenPictureDialog1.FileName;
```

```
    if Image1.Picture <> nil then begin
      ScrollBarHor.Max := Image1.Picture.Width — Panel1.Width;
      ScrollBarVert.Max := Image1.Picture.Height — Panel1.Height;
    end;
  end;
end;

procedure TForm1.ScrollBarHorChange(Sender: TObject);
begin
  Image1.Left := -ScrollBarHor.Position;
end;

procedure TForm1.ScrollBarVertChange(Sender: TObject);
begin
  Image1.Top := -ScrollBarVert.Position;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
end.
```

Прокрутка изображения осуществляется путем управления расположением левого верхнего угла компонента `Image1` через свойства `Left` и `Top`. Установка значений свойств выполняется в обработчиках события `OnChange` полос прокрутки.

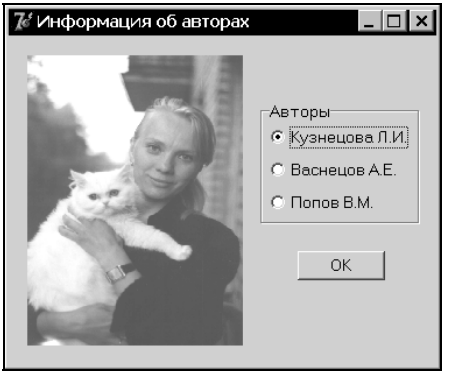


Рис. 10.8. Графическое приложение с выводом информации об авторах

Для выбора имени графического файла применено стандартное диалоговое окно `OpenPictureDialog1`. Вместо него можно использовать более общее диалоговое окно `OpenDialog1`, установив соответствующее значение для свойства `Filter`. Для установки фильтров графических файлов можно использовать редактор фильтров, а также специальную функцию `GraphicFilter`.

Рассмотрим еще один пример графического приложения. В форме расположены 3 компонента (`Image`, `RadioGroup`, `Button`) и подготовлены соответствующие обработчики событий. Вид формы показан на рис. 10.8.

Далее приводится код модуля `uImage2` формы приложения.

```
unit uImage2;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
         Image1: TImage;
        Button1: TButton;
    RadioGroup1: TRadioGroup;

    procedure Button1Click(Sender: TObject);
    procedure RadioGroup1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0: Image1.Picture.LoadFromFile('Kuznetsova.bmp');
    1: Image1.Picture.LoadFromFile('Vasnetsov.bmp');
    2: Image1.Picture.LoadFromFile('Popov.bmp');
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Image1.Stretch := True;
  Image1.Picture.LoadFromFile('Kuznetsova.bmp');
end;
end.
```

Группа переключателей `RadioGroup1` определяет автора, чью фотографию отображает компонент `Image`. При появлении окна отображается фотография Кузнецовой Л. И. Кнопка **OK** закрывает окно.

Для работы с картинками в формате JPEG в Delphi предназначен специальный класс `TJPEGImage`. Для использования этого класса и предоставляемых им возможностей в разделе `uses` следует подключить модуль `JPEG`.

Компонент *PaintBox*

Окно рисования (мольберт) представляет собой прямоугольную область, внутри которой можно выполнять операции рисования. Для работы с окном рисования в Delphi служит компонент `PaintBox` (расположен на странице **System** Палитры компонентов), основным свойством которого является свойство `Canvas` типа `TCanvas`. Окно рисования обычно используется, когда нужно ограничить поверхность рисования областью, размер которой меньше размера этой поверхности, а также для рисования на поверхности компонентов, не обладающих свойством `Canvas`.

Сам компонент `PaintBox` является невидимым, отображается только выводимое на нем изображение. В отличие от компонента `Image`, в компонент `PaintBox` нельзя загружать готовые изображения.

Компонент *ImageList*

Компонент `ImageList` (список графических изображений) предназначен для хранения набора графических изображений. Список представляет собой коллекцию однотипных изображений одинакового размера, на каждое из которых можно ссылаться по индексу. Списки изображений используются для эффективного управления множествами значков или битовых массивов. Сам список является *невизуальным компонентом* и на экране не отображается, содержащиеся в нем изображения также являются невидимыми. Видимыми они становятся только при отображении их каким-либо способом на поверхности какого-либо визуального компонента. Списки изображений удобно использовать для запоминания различного числа рисунков, которые при необходимости можно быстро отобразить.

В Delphi список графических изображений представлен компонентом `ImageList` (расположен на странице **Win32** Палитры компонентов), производным от класса `TCustomImageList` и являющимся контейнером для хранения нескольких изображений одинакового типа и размера. Свойства компонента `ImageList` определяют его характеристики и характеристики содержащихся в нем изображений, а методы позволяют оперировать ими. Элементами списка могут быть изображения типа "значок" (ico) или "битовый массив" (bmp).

Кроме того, в списке могут храниться *маски изображений*, которые определяют способ отображения при прорисовке. Размеры маски совпадают с размерами изображения. Нулевой бит маски указывает, что в этом месте изображения при его выводе на какой-либо поверхности будет нарисован бит с цветом фона. Ненулевой бит маски делает возможным вывод на прорисовываемой поверхности соответствующего ему бита изображения.

Элементы списка `ImageList` используются, например, для рисунков на кнопках `ToolButton` панели инструментов `ToolBar` или на ярлычках многостраничного блокнота `PageControl`. Эти компоненты имеют специальное свойство, значение которого указы-

вает имя списка графических изображений, из которого берутся рисунки. Таким свойством является, например, свойство `Images` типа `TCustomImageList`. Чтобы определить, какое именно изображение из заданного списка выводится в конкретном элементе, данный элемент имеет свойство, указывающее номер "своего" изображения. В частности, для кнопки `ToolButton` это свойство `ImageIndex` типа `Integer`.

Рисунок, выводимый на поверхности элемента, может зависеть от состояния этого элемента, например, вид глифа на кнопке панели инструментов зависит от того, активна кнопка или заблокирована. Если для элемента используется несколько рисунков, то они берутся из различных списков, указанных соответствующими свойствами того компонента, которому эти элементы принадлежат. Так, для панели инструментов `ToolBar` свойства `Images`, `HotImages` и `DisableImages` могут задавать три различных контейнера для изображений, которые отображаются на кнопках `ToolButton` в зависимости от их состояния.

Для компонентов, у которых нет свойства, указывающего список графических изображений, но которые требуют отображения в своей области рисунков из списка, программист самостоятельно выполняет вывод графики с помощью класса `TCanvas` и списка `ImageList`. Примеры программной прорисовки области компонента `ListBox` и организации вывода изображений из списка `ImageList` приводятся в *главе 3*.

Для работы с компонентом `ImageList` на этапе проектирования используется редактор (рис. 10.9), с помощью которого можно добавить в контейнер изображение или удалить его. Каждое изображение в компоненте `ImageList` имеет свой номер, отсчет начинается с нуля. Редактор позволяет перемещать отдельные изображения, изменяя их положения и, соответственно, их номера в контейнере. Кроме того, для отдельных изображений можно установить значения некоторых свойств, например, прозрачный (фоновый) цвет. Для каждого изображения может создаваться и запоминаться маска.

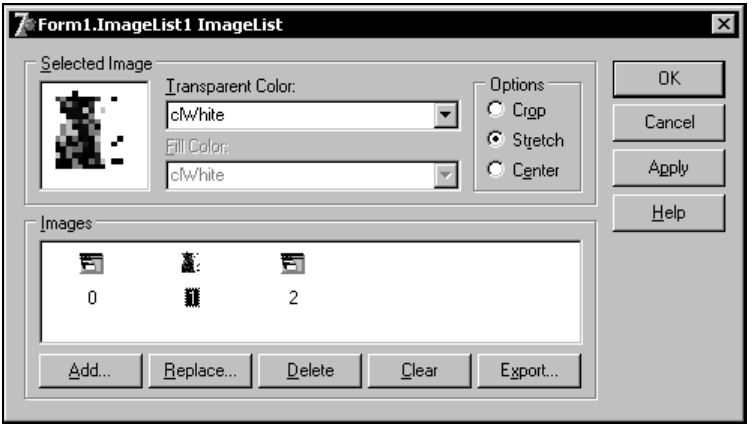


Рис. 10.9. Редактор компонента `ImageList`

Обычно при проектировании приложения в список графических изображений только загружается набор рисунков, а все остальные действия с этим списком реализуются уже в динамическом режиме, т. е. при выполнении приложения.

Все хранимые в списке изображения имеют одинаковый размер. По умолчанию он составляет 16×16 пикселов и устанавливается при создании компонента `ImageList`. При необходимости можно динамически *установить новый размер*. Для этого используется конструктор `CreateSize(AWidth, AHeight: Integer)`, в котором параметры `AWidth` и `AHeight` задают ширину и высоту изображения в пикселах.

Изменить размеры изображений в списке `ImageList` можно также, установив значения свойств `Width` и `Height` типа `Integer`, которые определяют ширину и высоту изображения и доступны для записи.

Замечание

При изменении значения любого из этих свойств все содержимое списка автоматически очищается. Поэтому значения этих свойств нужно устанавливать *до* загрузки изображений.

При использовании метода `CreateSize` подобного не происходит, и список `ImageList` сохраняет свое содержимое.

Для определения *числа изображений* в списке предназначено свойство `Count` типа `Integer`. При добавлении или удалении изображений значение этого свойства изменяется автоматически, также автоматически изменяются номера изображений в списке.

Загрузку изображений в контейнер можно выполнить не только с помощью редактора, но и динамически, в процессе выполнения программы. Для этого предназначены методы `Add`, `AddMasked`, `AddIcon`, `AddImages`, `Assign`, `Insert`, `InsertMasked`, `InsertIcon`, `Replace`, `ReplaceMasked` и `ReplaceIcon`.

Функция `Add(Image, Mask: TBitmap): Integer` *добавляет* в конец списка битовый массив и маску, заданные параметрами `Image` и `Mask` соответственно. В качестве результата возвращается значение индекса нового изображения в списке. Изображение и маска должны быть подготовлены до вызова данного метода.

Для *добавления изображений* удобно использовать функцию `AddMasked(Image: TBitmap; MaskColor: TColor): Integer`, которая формирует маску автоматически. Маска создается из пикселов изображения, указанного параметром `Image`, при этом учитывается цвет, заданный параметром `MaskColor`. В результате пикселы изображения, цвет которых совпадает с цветом `MaskColor`, рисуются прозрачными, а остальные выводятся своим цветом. Таким образом, параметр `MaskColor` задает фоновый цвет изображения. Если параметр `Image` содержит несколько изображений, они автоматически разделяются и добавляются к списку под разными номерами.

Приведем пример, в котором к списку `ImageList` добавляются новые рисунки.

```
procedure TForm1.Button1Click(Sender: TObject);
var bm: TBitmap;
begin
  // Создание экземпляра объекта TBitmap
  bm := TBitmap.Create;
  // Загрузка рисунка из файла
  bm.LoadFromFile('test.bmp');
  // Очистка списка ImageList1
  ImageList1.Clear;
```

```
  // Добавление к списку ImageList1 трех рисунков.
  // Фоновым является цвет левого верхнего пиксела изображения
  ImageList1.AddMasked(bm, bm.Canvas.Pixels[0, 0]);
  // Фоновым считается красный цвет
  ImageList1.AddMasked(bm, clRed);
  // Маска отсутствует
  ImageList1.Add(bm, nil);
  // Вывод на поверхности формы трех рисунков
  ImageList1.Draw(Form1.Canvas, 30, 20, 0);
  ImageList1.Draw(Form1.Canvas, 30, 50, 1);
  ImageList1.Draw(Form1.Canvas, 30, 80, 2);
  // Удаление экземпляра объекта TBitmap
  bm.Free;
end;
```

Растровый рисунок из файла test.bmp загружается в переменную `bm` типа `TBitmap`, откуда три раза добавляется к списку `ImageList1`. Операторы добавления изображения к списку отличаются маской. В первом случае в качестве прозрачного цвета взят цвет левого верхнего пиксела картинки. Во втором случае прозрачным является красный цвет, а в последнем варианте маска отсутствует. После формирования списка графических изображений его содержимое отображается на поверхности формы.

Функция `AddIcon(Image: TIcon): Integer` служит для *добавления* в конец списка графических изображений значка, определенного параметром `Image`. Копирование маски зависит от значения свойства `Masked` типа `Boolean`. Если свойство `Masked` имеет значение `True`, то значок копируется вместе с маской, если `False`, то без маски.

Процедуры `AddImages(Value: TCustomImageList)` и `Assign(Source: TPersistent)` предназначены для *копирования* содержимого *одного списка графических изображений* в другой. Процедура `AddImages` *добавляет* в конец списка содержимое другого списка, указанного параметром `Value`. Процедура `Assign` *заменяет* старое содержимое списка новым, взятым из источника, заданного параметром `Source`.

Методы `Insert`, `InsertMasked` и `InsertIcon` отличаются от соответствующих методов `Add`, `AddMasked` и `AddIcon` тем, что позволяют *задавать позицию* списка, в которую вставляется указанное изображение.

Методы `Replace`, `ReplaceMasked` и `ReplaceIcon` *заменяют изображение* и *маску*, находящиеся на указанной позиции в списке, заданными изображением и маской.

Для *перемещения изображения* внутри списка служит метод `Move`. Процедура `Move(CurIndex, NewIndex: Integer)` перемещает изображение с позиции, указанной параметром `CurIndex`, на новое место, номер которого задан параметром `NewIndex`.

Для *удаления изображений* из компонента `ImageList` предназначены методы `Clear` и `Delete`. Процедура `Clear` *удаляет все содержимое* списка, а процедура `Delete(Index: Integer)` *удаляет изображение*, позиция которого в списке задана параметром `Index`. Напомним еще раз, что полная очистка списка изображений происходит также при изменении значений его свойств `Width` или `Height`.

Как уже отмечалось, содержащиеся в списке ImageList изображения автоматически выводятся на поверхность элементов интерфейса, связанных с этим списком через соответствующие свойства, например, Images или HotImages. Кроме того, любой рисунок, находящийся в компоненте ImageList, можно отобразить на объекте типа TCanvas с помощью методов Draw и DrawOverlay.

Процедура Draw(Canvas: TCanvas; X, Y, Index: Integer; Enabled: Boolean=True) рисует на определяемой параметром Canvas поверхности *изображение*, номер которого задан параметром Index. Параметры X и Y указывают левый верхний угол, начиная с которого выводится рисунок. Параметр Enabled определяет доступность отображения, по умолчанию имеет значение True и обычно не указывается, при этом изображение доступно для обработки.

Для *управления способами прорисовки* изображения на указанной поверхности можно использовать такие свойства, как DrawingStyle, ImageType или BkColor.

Свойство DrawingStyle типа TDrawingStyle задает *стиль*, используемый для вывода изображения на поверхности рисования, и принимает следующие значения:

- dsFocused (цвета изображения на 25% смешиваются с системным цветом подсветки);
- dsSelected (цвета изображения на 50% смешиваются с системным цветом подсветки);
- dsNormal (изображение выводится с учетом цвета, заданного свойством BkColor: если оно имеет значение clNone, то рисование идет с использованием маски);
- dsTransparent (изображение рисуется с использованием маски, независимо от значения свойства BkColor).

Значения dsFocused и dsSelected влияют на те списки изображений, которые содержат маски.

Свойство BkColor типа TColor служит для определения *фонового цвета*, применяемого при выводе изображения. Значения свойства устанавливаются с помощью присваивания ему чисел или именованных констант clNone и clDefault. Константа clNone задает отсутствие фонового цвета, а константа clDefault указывает, что при прорисовке используется фоновый цвет списка графических образов. При выводе изображений, имеющих маски, их маскированные области рисуются цветом, значение которого указано свойством BkColor.

Объект, используемый при прорисовке содержимого списка ImageList, определяется свойством ImageType типа TImageType, принимающим следующие значения:

- itImage (метод Draw рисует указанное изображение);
- itMask (метод Draw рисует маску указанного изображения).

В отличие от метода Draw, процедура DrawOverlay позволяет выводить изображение на поверхности рисования с эффектом перекрытия цветов. Совместно с этим методом используется метод Overlay, определяющий маску.

Построение диаграмм

Диаграммы существенно облегчают восприятие различной числовой информации. Условно их можно разделить на две группы:

- индикаторы;
- сложные диаграммы и графики.

Для работы с диаграммами в Delphi имеются специальные компоненты.

Индикаторы

Индикаторы представляют собой диаграммы простейшего вида и предназначены для отображения в текстовом и графическом виде, например, хода выполнения длительных операций (форматирование дискеты, печать больших документов и т. п.). В Delphi индикаторы представлены компонентами `ProgressBar` и `Gauge`.

Компонент *ProgressBar*

Индикатор выполнения (компонент `ProgressBar`) расположен на странице **Win32** Палитры компонентов и представляет собой графическую полосу (рис. 10.10), показывающую ход выполнения продолжительной операции. По мере выполнения операции эта полоса заполняется слева направо цветным полем, причем длина поля соответствует проценту выполнения отображаемой операции. Стиль компонента `ProgressBar` согласован с современным интерфейсом Windows.

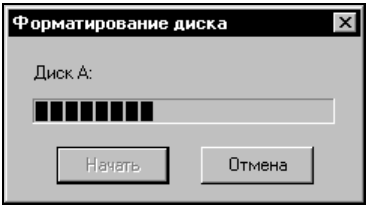


Рис. 10.10. Использование компонента `ProgressBar`

Замечание

Компонент `ProgressBar` не имеет свойств, позволяющих управлять цветами фона, и полей индикации. Цвет фона графической полосы индикатора устанавливается равным цвету компонента-контейнера, в котором индикатор размещается. Цветное поле имеет синий цвет.

К основным свойствам компонента `ProgressBar` относятся следующие свойства типа `Integer`:

- `Min` и `Max` (минимальное и максимальное значения диапазона индикатора). По умолчанию значение свойства `Min` равно `0`, а значение свойства `Max` — `100`;
- `Position` (позиция индикатора, т. е. текущий объем выполненной операции). Объем выполненной операции рассчитывается относительно диапазона, заданного значениями свойств `Min` и `Max`. Например, если значение свойства `Min` равно `0`, а значение

свойства Min — 80, то при значении свойства Position, равном 20, индикатор хода работ укажет, что операция выполнена на 25 процентов;

- Step (величина, на которую возрастает значение свойства Position при каждом наращивании индикатора). По умолчанию свойство имеет значение 10.

Позицией индикатора ProgressBar можно управлять, устанавливая значение свойства Position, например, в инструкции присваивания: ProgressBar.Position := 35;.

В случае выхода значения свойства Position за границы заданного диапазона возможны два варианта: индикатор показывает 100% выполнения объема работ, если значение Position больше значения свойства Max, и 0%, если значение свойства Position меньше значения свойства Min.

Изменить позицию элемента ProgressBar можно также с помощью методов StepIt и StepBy. Процедура StepIt не имеет параметров и *увеличивает* позицию индикатора на *значение свойства* Step. Процедура StepBy(Delta: Integer) *увеличивает* позицию индикатора на *значение параметра* Delta.

Например, инструкция ProgressBar1.StepBy(13); увеличивает позицию индикатора ProgressBar1 на 13.

Обычно индикаторы хода работ сопровождаются текстовой надписью, отображающей объем выполненной операции. Компонент ProgressBar не поддерживает таких свойств, как Text или Caption, поэтому текстовую надпись к нему необходимо добавлять отдельно, например, с помощью компонента Label.

Компонент *Gauge*

Компонент Gauge расположен на странице **Samples** Палитры компонентов и предназначен для отображения простейших диаграмм. Он представляет собой *индикатор*, показывающий значение какого-либо параметра в процентном соотношении. Последнее является важным достоинством компонента Gauge, хотя его стиль и не соответствует современному интерфейсу Windows.

Важнейшим свойством компонента Gauge является свойство Kind, которое определяет *вид диаграммы* и принимает следующие значения:

- gkHorizontalBar (горизонтальный прямоугольник);
- gkVerticalBar (вертикальный прямоугольник);
- gkPie (сектор);
- gkNeedle (спидометр);
- gkText (только текст).

На рис. 10.11 показаны различные формы диаграмм, отображаемых с помощью компонента Gauge.

К числу основных свойств компонента Gauge, наряду с Kind, относятся следующие:

- BorderStyle типа TBorderStyle (вид рамки); принимает одно из двух значений:
  - bsSingle (тонкая линия);
  - bsNone (рамки нет);

- `ForeColor` типа `TColor` (цвет индикатора). По умолчанию используется черный цвет (значение `clBlack`);
- `BackColor` типа `TColor` (цвет для прорисовки части диаграммы, свободной от индикатора). По умолчанию используется белый цвет (значение `clWhite`);
- `MinValue` типа `Integer` (минимальное значение отображаемого параметра, на диаграмме принимается за 0%);
- `MaxValue` типа `Integer` (максимальное значение отображаемого параметра, на диаграмме принимается за 100%);
- `Progress` типа `Integer` (текущее значение отображаемого параметра в процентах);
- `ShowText` типа `Boolean` (управляет отображением текста на диаграмме). Если свойство `ShowText` имеет значение `True` (по умолчанию), то вместе с графиком отображается текст. Если свойство имеет значение `False`, то текст на диаграмме не виден.

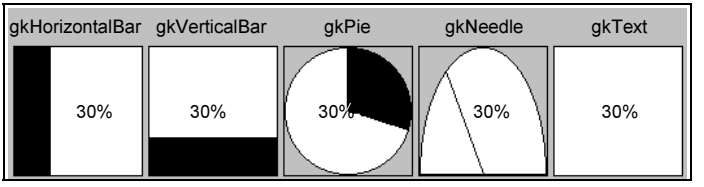


Рис. 10.11. Формы диаграмм, формируемых компонентом `Gauge`

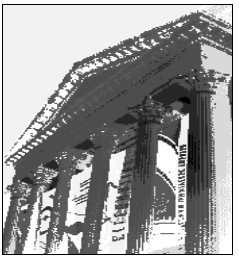
Далее приводится пример текста процедуры, использующей индикатор `Gauge`.

```
procedure TForm1.Button1Click(Sender: TObject);
var MemoryTick: longint;
begin
  Gauge1.Kind        := gkPie;
  Gauge1.BorderStyle := bsNone;
  Gauge1.ForeColor   := clRed;
  Gauge1.BackColor   := clWhite;
  Gauge1.MinValue    := 0;
  Gauge1.MaxValue    := 60;
  Gauge1.ShowText    := False;
  MemoryTick         := GetTickCount;
  while GetTickCount <= MemoryTick + 60000 do
      Gauge1.Progress := (GetTickCount — MemoryTick) div 1000;
end;
```

В форме рисуется круговая диаграмма, в течение минуты отображающая секундомер. Для отсчета времени используется API-функция `GetTickCount`, возвращающая от встроенных часов компьютера количество тиков, прошедших с момента запуска Windows, при этом 1 секунда равна 1000 тикам. Отметим, что недостатком данного кода является то, что до истечения одной минуты невозможно выйти из цикла `while...do`. В реальных программах подобных ситуаций следует избегать.

Компонент *Chart* (диаграмма)

Компонент-диаграмма `Chart` предназначен для работы со сложными диаграммами различных типов, в том числе с объемными. Работа с компонентом-диаграммой рассмотрена в *главе 18*, посвященной визуальным компонентам для работы с базами данных, на примере аналогичного компонента `DBChart`. В дополнение к свойствам компонента `Chart` компонент-диаграмма `DBChart` позволяет строить графики по информации из таблиц баз данных.



ГЛАВА 11

Использование средств мультимедиа

Мультимедийные технологии позволяют повысить качество программ, придать им более профессиональный и привлекательный для пользователя вид. Из разнообразных средств мультимедиа наиболее интересны аудио- и видеовозможности компьютера.

В языке Delphi нет предназначенных для работы со звуком процедур типа `Sound` или `NoSound`, имеющихся в языках Turbo Pascal и Borland Pascal. Для использования мультимедийных возможностей и средств компьютера в Delphi служат специальные компоненты `Animate` и `MediaPlayer`.

Компонент `Animate`, расположенный на странице **Win32** Палитры компонентов, предназначен для создания простой *анимации* и позволяет проигрывать файлы формата AVI (Audio-Video Interleaved — "перемежаемые" изображение и звук). Компонент `MediaPlayer` расположен на странице **System** Палитры компонентов и представляет собой сложный *многофункциональный элемент*, который обеспечивает воспроизведение аудио- и видеофайлов, а также управление соответствующими устройствами. Во многом эти компоненты похожи друг на друга и имеют одинаковые или подобные составляющие элементы.

В простейших случаях для генерации звукового сигнала можно использовать процедуру `Beep` модуля `SysUtils`. Эта процедура вызывает одноименную API-функцию `Beep`, издающую стандартный системный звук с помощью встроенного динамика.

Рассмотрим следующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  try
    Edit1.Text := IntToStr(StrToInt(Edit1.Text) + 1);
  except
    Beep;
    MessageDlg('Ошибка в поле Edit1!', mtError, [mbOK], 0);
  end;
end;
```

Редактор `Edit1` содержит целое число. При нажатии кнопки `Button1` это число увеличивается на единицу. В случае возникновения какого-либо исключения, например ошибки преобразования, выдается звуковой сигнал и выводится сообщение.

Для получения звука можно использовать также API-функцию `MessageBeep(uType: UINT): Boolean`, генерирующую стандартный системный звук, тип которого указан параметром `uType`. Параметр `uType` задается в виде шестнадцатеричного числа или с помощью именованных констант: например, шестнадцатеричное число `$FFFFFFFF` соответствует стандартному системному звуку, а именованная константа `MB_OK` — системному звуку по умолчанию.

При успешном вызове функции `MessageBeep` воспроизведение звука осуществляется асинхронно (параллельно) с дальнейшим выполнением приложения, а в качестве результата возвращается значение `True`.

Выдаваемые звуковые сигналы зависят от настроек, выполненных в Панели управления Windows. Так, если невозможно воспроизвести звук заданного типа, то воспроизводится системный звук по умолчанию.

Воспроизведение видеоклипов

Видеоклип представляет собой файл в формате AVI. Этот файл содержит последовательность отдельных изображений (кадров), при отображении которых создается эффект движения изображения. Наряду с изображением, avi-файлы могут содержать звук. Для воспроизведения видеоклипов подходит любой из компонентов `Animate` или `MediaPlayer`.

Компонент `Animate` позволяет проигрывать avi-файлы, а также отображать стандартную анимацию, используемую Windows. Однако avi-файлы, воспроизводимые этим компонентом, имеют определенные ограничения:

- они не должны содержать звука;
- информация в них не должна быть сжатой;
- размер файла не должен превышать 64 Кбайт.

Для задания воспроизводимого видеоклипа используются свойства `FileName` типа `TFileName` и `CommonAVI` типа `TCommonAVI`. Одновременно можно применять одно из этих свойств: так, если в качестве значения свойства `FileName` указывается avi-файл, существующий на диске, то свойство `CommonAVI` автоматически устанавливается в значение `aviNone`. Свойство `CommonAVI` позволяет выбрать один из стандартных клипов Windows, обеспечиваемых в Shell32.dll, и принимает следующие значения:

- `aviNone` (отсутствие стандартной анимации);
- `aviCopyFile` (копирование файла);
- `aviCopyFiles` (копирование файлов);
- `aviDeleteFile` (удаление файла);
- `aviEmptyRecycle` (очистка Корзины);
- `aviFindComputer` (поиск компьютера);
- `aviFindFile` (поиск файла);
- `aviFindFolder` (поиск папки);
- `aviRecycleFile` (перемещение файла в корзину).

При установке свойства `CommonAVI` в отличное от `aviNone` значение свойство `FileName` автоматически сбрасывается, принимая в качестве значения пустую строку.

Для задания видеоклипа также можно использовать свойства `ResHandle` типа `THandle` и `ResID` типа `Integer`, представляющие собой альтернативы свойствам `CommonAVI` и `FileName`. Значение свойства `ResHandle` *задает ссылку на модуль*, который содержит изображение в виде ресурса, а значение свойства `ResID` *указывает номер ресурса* в этом модуле. В случае успешной загрузки видеоклипа из ресурса свойство `CommonAVI` автоматически устанавливается в значение `aviNone`, а значение свойства `FileName` сбрасывается в пустую строку.

После выбора видеоклипа свойства `FrameCount`, `FrameHeight` и `FrameWidth` типа `Integer` определяют *число кадров*, а также *высоту* и *ширину кадров* (в пикселах) соответственно. Все эти свойства доступны только во время выполнения и только для чтения.

По умолчанию размеры компонента `Animate` автоматически подстраиваются под размеры кадров видеоклипа, это определяет значение `True` свойства `AutoSize` типа `Boolean`. Если данное свойство установить в значение `False`, то компонент `Animate` свои размеры не изменяет, при этом возможно отсечение части кадра изображения, если его размеры превышают размеры компонента `Animate`.

Воспроизведение видеоклипа начинается при установке свойства `Active` типа `Boolean` в значение `True`. *Начальный* и *конечный кадры* задают диапазон воспроизведения и определяются, соответственно, значениями свойств `StartFrame` и `StopFrame` типа `SmallInt`. По умолчанию свойство `StartFrame` указывает на первый кадр анимации и имеет значение `1`, а свойство `StartFrame` — на последний кадр, и его значение равно значению свойства `FrameCount`.

Свойство `Repetitions` типа `Integer` определяет *число повторений* воспроизведения видеоклипа. По умолчанию оно имеет значение `0`, и видеоклип проигрывается неограниченное число раз до тех пор, пока процесс воспроизведения не будет остановлен.

Для *принудительной* (до истечения заданного числа повторений) *остановки воспроизведения* видеоклипа свойство `Active` следует установить в значение `False`.

Свойство `Center` типа `Boolean` задает, будет ли изображение выводиться в центре компонента `Animate` или в его левом верхнем углу. По умолчанию это свойство имеет значение `True`, и видеоклип проигрывается в центре области компонента `Animate`.

Для запуска и остановки воспроизведения клипов можно использовать также методы `Play`, `Stop` и `Reset`. Процедура `Play(FromFrame, ToFrame: Word; Count: Integer)` *проигрывает видеоклип*, начиная с кадра, заданного параметром `FromFrame`, и заканчивая кадром, заданным параметром `ToFrame`. Параметр `Count` определяет число повторений. Таким образом, эта процедура позволяет одновременно управлять свойствами `StartFrame`, `StopFrame` и `Repetitions`, задавая для них требуемые при воспроизведении значения, а также устанавливает свойство `Active` в значение `True`.

Процедура `Stop` *прерывает воспроизведение* видеоклипа и устанавливает свойство `Active` в значение `False`. Процедура `Reset`, кроме того, дополнительно сбрасывает свойства `StartFrame` и `StopFrame`, устанавливая для них значения по умолчанию.

Свойство `Open` типа `Boolean` доступно при выполнении программы и позволяет определить, *готов ли компонент* `Animate` к воспроизведению. Если выбор и загрузка видео-

клипа проходят успешно, свойство `Open` автоматически устанавливается в значение `True`, и компонент `Animate` можно открыть — проиграть анимацию. Если загрузить видеоклип не удается, то это свойство получает значение `False`. При необходимости программист может сам устанавливать свойство `Open` в значение `False`, отключая этим компонент `Animate`.

Рассмотрим пример приложения, реализующего просмотр стандартной анимации Windows (рис. 11.1).

Рис. 11.1. Стандартная анимация Windows

С помощью группы переключателей **Действие** (`RadioGroup1`) выбирается вид (эффект) анимации, после чего видеоклип можно просмотреть, нажав кнопку **Старт**. Видеоклип воспроизводится неограниченное число раз с первого кадра по последний, для его прерывания нужно нажать кнопку **Стоп**. Кнопки **Старт** и **Стоп** при нажатии взаимно блокируются. При выборе в группе **Действие** анимации другого вида воспроизведение, если оно было запущено, останавливается. Надпись `Label1`, расположенная над компонентом `Animate1`, отображает название выбранного вида анимации.

В листинге 11.1 приводится исходный код модуля `uAnimate` формы приложения.

Листинг 11.1. Пример просмотра стандартной анимации Windows

```
unit uAnimate;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    btnStart: TButton;
     btnExit: TButton;
     btnStop: TButton;
    Animate1: TAnimate;
      Label1: TLabel;
 RadioGroup1: TRadioGroup;
```

```
    procedure btnExitClick(Sender: TObject);
    procedure btnStartClick(Sender: TObject);
    procedure btnStopClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure RadioGroup1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
const aviType: array[0..8] of TCommonAVI =
         (aviNone, aviCopyFile, aviCopyFiles,
          aviDeleteFile, aviEmptyRecycle, aviFindComputer,
          aviFindFile, aviFindFolder, aviRecycleFile);
var Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  RadioGroup1.ItemIndex := 0;
  RadioGroup1Click(Sender);
end;

procedure TForm1.btnStartClick(Sender: TObject);
begin
  Animate1.Play(1, Animate1.FrameCount, 0);
  btnStart.Enabled := False;
  btnStop.Enabled  := True;
end;

procedure TForm1.btnStopClick(Sender: TObject);
begin
  Animate1.Stop;
  btnStart.Enabled := True;
  btnStop.Enabled  := False;
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  Animate1.CommonAVI := aviType[RadioGroup1.ItemIndex];
  Label1.Caption := RadioGroup1.Items[RadioGroup1.ItemIndex];
  btnStop.Click;
  if RadioGroup1.ItemIndex = 0
    then btnStart.Enabled := False
    else btnStart.Enabled := True;
end;

procedure TForm1.btnExitClick(Sender: TObject);
begin
  Close;
end;
end.
```

Как уже говорилось, компонент Animate обеспечивает воспроизведение только простых avi-файлов. Для проигрывания больших файлов предназначен компонент MediaPlayer, предоставляющий гораздо более богатые мультимедийные возможности *(см. разд. "Управление мультимедийными устройствами" далее в этой главе)*.

Часто компонент Animate используется при *создании панелей инструментов* ToolBar или CoolBar для *добавления* в *них анимационных значков*, которые оживляют форму и служат для индикации того, что программа выполняет ту или иную обработку данных. Воспроизведение изображения при этом осуществляется, например, при нажатии кнопки панели инструментов или по истечении заданного интервала времени.

Рассмотрим пример такой "анимированной" панели инструментов.

Форма Form2 содержит панель инструментов CoolBar1 с двумя полосами (рис. 11.2). В первой полосе расположена панель ToolBar1 с четырьмя элементами типа TToolButton (три кнопки и разделитель) и компонентом Animate1. Во второй полосе находится компонент Animate2.

В листинге 11.2 приводится исходный код модуля u2Animate формы Form2.

Листинг 11.2. Пример "анимированной" панели инструментов

```
unit u2Animate;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ToolWin, ComCtrls, ExtCtrls;
type
  TForm2 = class(TForm)
       CoolBar1: TCoolBar;
       ToolBar1: TToolBar;
    ToolButton1: TToolButton;
    ToolButton2: TToolButton;
    ToolButton3: TToolButton;
    ToolButton4: TToolButton;
         Timer1: TTimer;
       Animate1: TAnimate;
       Animate2: TAnimate;

    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure ToolButton1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var Form2: TForm2;
implementation
{$R *.DFM}
```

```
procedure TForm2.FormCreate(Sender: TObject);
begin
  Timer1.Interval := 5000;
  try
    Animate1.FileName := 'anim1.avi';
  except
    ShowMessage('Не загружен первый файл анимации!');
  end;
  try
    Animate2.FileName := 'anim2.avi';
  except
    ShowMessage('Не загружен второй файл анимации!');
  end;
end;

procedure TForm2.ToolButton1Click(Sender: TObject);
begin
  if not Animate1.Open then exit;
  Animate1.Play(1, Animate1.FrameCount, 1);
end;

procedure TForm2.Timer1Timer(Sender: TObject);
begin
  if not Animate2.Open then exit;
  Animate2.Play(1, Animate2.FrameCount, 1);
end;
end.
```

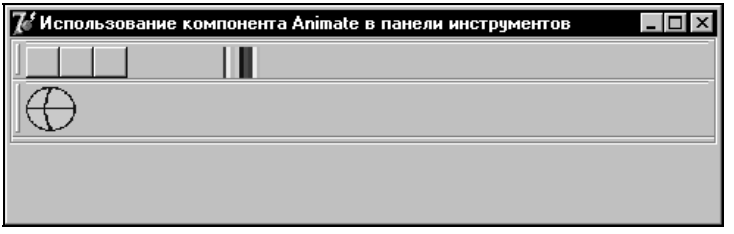


Рис. 11.2. Компонент `Animate` в панели инструментов

Нажатие кнопки `ToolButton1` вызывает запуск анимации в компоненте `Animate1` (движущиеся цветные полосы).

Компонент `Animate2` оживляет программу, с этой целью каждые пять секунд он воспроизводит видеоклип — вращающийся шар. Запуск этого клипа выполняется в обработчике события `OnTimer` компонента `Timer1`, для которого установлен интервал, равный 5000.

Загрузка видеофайлов производится при создании формы. Если при этом произойдет какая-либо ошибка, то соответствующий анимационный эффект будет отсутствовать.

Управление мультимедийными устройствами

Многие мультимедийные устройства имеют так называемый MCI-интерфейс (Media Control Interface — интерфейс управления мультимедиа). С помощью MCI-интерфейса осуществляется управление объектами типа аудио- и видеофайлов, а также устройствами мультимедиа, такими как звуковые карты, карты обработки видеосигналов, видеомагнитофоны, лазерные проигрыватели. В Delphi работу с подобными устройствами осуществляет компонент `MediaPlayer` (мультимедийный проигрыватель), который обеспечивает стыковку Delphi-программы с драйверами MCI.

Мультимедийный проигрыватель `MediaPlayer`, в отличие от компонента `Animate`, является многофункциональным элементом управления и предоставляет программисту большой набор свойств и методов, позволяющих манипулировать файлами и устройствами мультимедиа, поддерживать воспроизведение, запись и перемещение между отдельными фонограммами (дорожками, записями), а также идентифицировать подключенные устройства.

Визуально компонент `MediaPlayer` представляет собой набор кнопок (рис. 11.3), с помощью которых формируется панель управления. Эту панель можно использовать для управления различными аппаратными и программными средствами мультимедиа.

Рис. 11.3. Вид компонента `MediaPlayer`

Компонент `MediaPlayer` содержит следующие кнопки (рис. 11.3, слева направо):

- **Play** (воспроизведение);
- **Pause** (пауза);
- **Stop** (остановка);
- **Next** (переход к следующей фонограмме (дорожке)); если фонограмма только одна, выполняется переход в ее конец;
- **Prev** (переход к предыдущей фонограмме); если фонограмма только одна, выполняется переход в ее начало;
- **Step** (переход на несколько кадров вперед);
- **Back** (возврат на несколько кадров назад);
- **Record** (включение режима записи);
- **Eject** (извлечение носителя).

Видимостью и *доступностью* кнопок мультимедийного проигрывателя можно управлять с помощью свойств `VisibleButtons`, `EnabledButtons` и `ColoredButtons`.

Свойство `VisibleButtons` типа `TButtonSet` определяет, какие кнопки в компоненте `MediaPlayer` являются *видимыми*. По умолчанию видимы все кнопки, что не всегда удобно. Например, при воспроизведении звуковых файлов формата WAV кнопки **Step**,

Back и **Eject** не нужны, и лучше сделать их невидимыми. Тип `TButtonSet` содержит значения, соответствующие кнопкам проигрывателя, и описан так:

```
type TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep, btBack,
                   btRecord, btEject);
TButtonSet = set of TMPBtnType;
```

Обычно набор видимых кнопок мультимедийного проигрывателя устанавливается через Инспектор объектов, но можно задать его и при выполнении программы. Например, если требуется отображать только кнопки воспроизведения, паузы и остановки, то достаточно следующей инструкции:

```
MediaPlayer1.VisibleButtons := [btPlay,btPause,btStop];
```

Свойство `EnabledButtons` типа `TButtonSet` определяет, какие кнопки *доступны* в элементе мультимедиа. Доступная кнопка выделяется цветом и может быть нажата. *Недоступная* (запрещенная) кнопка имеет светло-серый цвет и не реагирует на нажатие. По умолчанию доступны все кнопки, однако мультимедиапроигрыватель в зависимости от ситуации может сам *управлять доступностью* своих кнопок, что определяется свойством `AutoEnable` типа `Boolean`. Если это свойство имеет значение `True` (по умолчанию), то проигрыватель автоматически изменяет доступность отдельных кнопок в зависимости от устройства, которым он манипулирует, и режима, в котором он находится. Например, если воспроизводится звуковой файл, то блокируется кнопка **Eject**. Таким образом, если свойство `AutoEnable` установлено в значение `True`, то программист может управлять доступностью только тех кнопок проигрывателя, которые не заблокированы автоматически самим проигрывателем.

Свойство `ColoredButtons` типа `TButtonSet` определяет, какие кнопки проигрывателя являются *цветными*. Нецветная кнопка отображается в оттенках серого. По умолчанию цветными являются все кнопки.

При выполнении программы пользователь нажимает кнопки с помощью мыши или клавиатуры. В случае использования клавиатуры нажатие выбранной кнопки выполняется клавишей <Пробел>, а перемещение между кнопками проигрывателя производится с помощью клавиш <←> и <→>. При нажатии кнопки вызывается соответствующий метод, выполняющий требуемые действия, например, метод `Stop` осуществляет остановку воспроизведения.

Название и назначение большинства таких методов (`Play`, `Pause`, `Stop`, `Next`, `Step`, `Back` и `Eject`) совпадают с названием и назначением вызывающих их кнопок. Исключениями являются метод `StartRecording`, выполняемый при нажатии кнопки **Record**, и метод `Previous`, которому соответствует кнопка **Prev**. Обычно при управлении компонентом `MediaPlayer` программист вызывает эти методы самостоятельно.

В процессе работы мультимедийный проигрыватель связан с конкретным файлом на внешнем носителе, чаще всего на лазерном или магнитном диске. Этот файл открыт для воспроизведения и/или записи, а *имя файла* определяет свойство `FileName` типа `String`.

В качестве примера рассмотрим процедуру, в которой создание формы сопровождается звуковыми эффектами:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MediaPlayer1.Visible    := False;
  MediaPlayer1.DeviceType := dtAutoSelect;
  MediaPlayer1.FileName   := 'start.wav';
  try
    if not MediaPlayer1.AutoOpen then MediaPlayer1.Open;
    MediaPlayer1.Play;
  except
    exit;
  end;
end;
```

При создании формы `Form1` воспроизводится звуковой файл start.wav. При возникновении исключения, связанного, например, с ошибкой чтения файла, звук отсутствует и предупреждающее сообщение не выдается.

В общем случае компонент `MediaPlayer1` можно использовать для озвучивания других событий и различных действий пользователя, например, нажатия кнопок.

В каждый момент времени мультимедийный проигрыватель может управлять только одним *устройством*, задаваемым в свойстве `DeviceType` типа `TMPDeviceTypes`. Это свойство принимает следующие значения:

- `dtAutoSelect` (автоматическое распознавание типа устройства);
- `dtAVIVideo` (avi-файл);
- `dtCDAudio` (аудиокомпакт-диск);
- `dtDAT` (цифровая аудиолента);
- `dtDigitalVideo` (цифровое видео); допускаются avi-, mpg- и mov-файлы;
- `dtMMMovie` (mov-фильм);
- `dtOther` (другое устройство);
- `dtOverlay` (аналоговое видео);
- `dtScanner` (сканер);
- `dtSequencer` (midi-файл);
- `dtVCR` (видеокассета);
- `dtVideodisc` (видеокомпакт-диск);
- `dtWaveAudio` (wav-файл).

По умолчанию установлено значение `dtAutoSelect`, что означает автоматическое распознавание типа открытого устройства в зависимости от расширения имени файла, указанного в свойстве `FileName`. Напомним, что расширения имен файлов связаны с конкретными устройствами и средствами Windows.

Перед использованием устройства его следует открыть, поскольку большинство методов, например `Play` и `StartRecording`, можно вызывать только после открытия устройства. Открытие устройства выполняется вызовом метода `Open`. Если необходимо автоматическое открытие устройства, то свойство `AutoOpen` типа `Boolean` следует установить

в значение `True`. По умолчанию это свойство имеет значение `False`, и при создании формы устройство, связанное с компонентом `MediaPlayer`, автоматически не открывается.

После того как устройство открыто, свойство `DeviceID` типа `Word` проигрывателя определяет идентификатор этого устройства. Если открытых устройств нет, то свойство `DeviceID` имеет значение ноль.

Например, в приводимой далее процедуре открывается проигрыватель аудиокомпакт-дисков:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  MediaPlayer1.DeviceType := dtCDAudio;
  MediaPlayer1.Open;
end;
```

Когда использование мультимедийного устройства прекращается, его нужно закрыть, вызвав метод `Close`. Этот метод вызывается автоматически при завершении работы с приложением и при удалении формы, в которой находится компонент `MediaPlayer`.

Свойство `Capabilities` типа `TMPDevCapsSet` позволяет определить возможности выбранного и открытого мультимедийного устройства. Это свойство может принимать набор следующих значений, устанавливающих доступность соответствующих операций:

- `mpCanEject` (извлечение носителя);
- `mpCanPlay` (воспроизведение);
- `mpCanRecord` (запись на носитель);
- `mpCanStep` (переход вперед или назад на определенное число кадров);
- `mpUsesWindow` (использование окна для вывода изображения).

После открытия устройства с помощью свойства `Tracks` типа `Longint` можно получить информацию о *числе фонограмм* (дорожек); если устройство не поддерживает дорожки, то значение этого свойства не определено. Свойство `TrackLength[TrackNum: Integer]` типа `Longint` содержит *длину фонограммы* с индексом `TrackNum` (отсчет начинается с единицы). Длина дорожки указывается в формате времени, который определен свойством `TimeFormat`.

Свойство `TimeFormat` типа `TMPTimeFormats` задает *формат значений* свойств, связанных со временем. Оно влияет на способ интерпретации и отображения значений таких свойств, как `TrackLength`, `Length`, `StartPos`, `EndPos`, `Position`. Каждое устройство поддерживает определенные для него форматы времени. Свойство `TimeFormat` принимает следующие значения:

- `tfMilliseconds` (целое четырехбайтовое число, определяет количество миллисекунд);
- `tfHMS` (числа (определяют часы, минуты и секунды), размещенные побайтно, начиная с младшего байта, в четырехбайтовом целом; старший байт не учитывается);
- `tfMSF` (числа (определяют минуты, секунды и количество кадров), размещенные побайтно, начиная с младшего байта, в четырехбайтовом целом; старший байт не используется);

- `tfFrames` (целое четырехбайтовое число, содержит количество кадров);
- `tfSMPTE24`, `tfSMPTE25`, `tfSMPTE30` и `tfSMPTE30Drop` — каждое из первых трех значений определяет часы, минуты, секунды и число блоков по 24, 25 и 30 кадров в секунду соответственно; последнее значение определяет часы, минуты, секунды и число пропущенных блоков по 30 кадров в секунду;
- `tfBytes` (четырехбайтовое целое, содержит количество байтов);
- `tfSamples` (четырехбайтовое целое, содержит количество условных блоков информации);
- `tfTMSF` (числа (определяют дорожки, минуты, секунды и количество кадров), размещенные побайтно, начиная с младшего байта, в четырехбайтовом целом).

Свойство `Position` типа `Longint` доступно для чтения при выполнении программы и указывает *текущую позицию* открытого устройства в установленном формате времени. При первоначальном открытии носителя значение свойства `Position` устанавливается в начало или в позицию, определенную свойством `StartPos`.

Свойства `StartPos` и `EndPos` типа `Longint` определяют, соответственно, *начальную* и *конечную позиции*, в пределах которых осуществляется воспроизведение. По умолчанию значение свойства `StartPos` устанавливается в начальную позицию носителя, а значение свойства `EndPos` — в конечную. Значения указываются в текущем формате времени. Оба свойства доступны для записи во время выполнения программы. Установки, сделанные для свойств `StartPos` и `EndPos`, действуют только на следующий вызов методов `Play` или `StartRecording`, после чего для них снова устанавливаются значения по умолчанию. При необходимости повторного вызова желаемые значения начала и конца воспроизведения должны устанавливаться заново.

В следующей процедуре проигрывается звуковой файл test.wav:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
     FileName := 'test.wav';
     Open;
     TimeFormat := tfMilliseconds;
     StartPos   := 10000;
     EndPos     := 25000;
     Play;
  end;
end;
```

Воспроизведение файла осуществляется с 10-й по 25-ю секунду записи. Время отсчитывается в миллисекундах.

При воспроизведении компонентом `MediaPlayer` видеоизображений свойство `Display` типа `TWinControl` задает *имя оконного элемента управления*, в котором осуществляется отображение. По умолчанию свойство `Display` имеет значение `nil`, и для отображения создается собственное окно. Программист может задать для вывода свое окно или какой-либо компонент, например, панель. Свойство `DisplayRect` типа `TRect` определяет *прямоугольную область*, используемую для вывода изображения, значение этого свойства следует устанавливать после открытия проигрывателя. Оба свойства доступны во

время выполнения приложения. Вывод в окно поддерживают устройства таких типов, как `dtAVIVideo`, `dtDigitalVideo`, `dtOverlay`, `dtVCR`, `dtVideodisc`.

Пример отображения видеофайла:

```
procedure TForm1. FormCreate(Sender: TObject);
begin
  with MediaPlayer1 do begin
     DeviceType := dtAVIVideo;
     FileName := ExtractFilePath(Application.ExeName) + 'move.avi';
     Open;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
     Display := Panel1;
     Play;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
     Display := nil;
     Play;
  end;
end;
```

При нажатии кнопки `Button1` видеофайл move.avi отображается на поверхности компонента `Panel1`, а при нажатии кнопки `Button2` для вывода изображения используется отдельное окно. Загрузка файла и установка типа устройства выполняются при создании формы.

Текущий *режим* мультимедийного проигрывателя указывает свойство `Mode` типа `TMPModes`, принимающее следующие значения:

- `mpNotReady` (устройство не готово);
- `mpStopped` (текущая операция завершена);
- `mpPlaying` (выполняется воспроизведение);
- `mpRecording` (выполняется запись);
- `mpSeeking` (выполняется поиск файла на носителе);
- `mpPaused` (операция приостановлена);
- `mpOpen` (устройство открыто).

Свойство `Error` типа `Longint` содержит *код ошибки*, возвращенный при выполнении устройством последней операции. Если последняя операция завершилась успешно, то в `Error` записывается `0`. Получить *описание ошибки* можно с помощью свойства `ErrorMessage` типа `String`, которое содержит строку, поясняющую последнюю возникшую ошибку.

Рассмотрим пример:

```
procedure TForm1.Button6Click(Sender: TObject);
begin
  MediaPlayer1.Next;
  if MediaPlayer1.Error = 0
    then MediaPlayer1.Play
    else MessageDlg(MediaPlayer1.ErrorMessage, mtError, [mbOK], 0);
end;
```

В компоненте `MediaPlayer1` выполняется переход на следующую фонограмму. Если операция выполнена успешно и код ошибки равен нулю, то фонограмма воспроизводится. В противном случае выдается сообщение об ошибке.

Для обработки ошибок, возникающих при работе с компонентом `MediaPlayer`, также могут использоваться рассмотренные ранее конструкции `try..except` и `try..fynally`.

При перемещении носителя с помощью методов `Step` и `Back` свойство `Frames` типа `Longint` определяет *количество кадров*, на которое изменяется текущая позиция. Свойство доступно во время выполнения программы, значение по умолчанию составляет 10% от длины текущего носителя.

Свойство `AutoRewind` типа `Boolean` определяет, выполняет ли мультимедийное устройство *перемотку* перед воспроизведением или записью. Если свойство `AutoRewind` имеет значение `True` (по умолчанию), то текущая позиция устройства перемещается в начало. В противном случае воспроизведение или запись начинается с текущей позиции. Свойство `AutoRewind` не действует, если в устройстве используются дорожки или если программист присвоил какие-либо значения свойствам `StartPos` и `EndPos`.

Если открытое устройство поддерживает режим записи, то его носитель можно с помощью метода `Save` *сохранить в файле*, заданном свойством `FileName`. Для устройств типа CD-Audio метод игнорируется.

После своего вызова каждый метод компонента `MediaPlayer` может *возвращать управление приложению* различными способами, определяемыми названием метода и значением свойства `Wait` типа `Boolean`, доступным во время выполнения. Если свойство `Wait` имеет значение `False`, то приложение продолжает свою работу обычным порядком, а если значение `True`, то приложение ожидает завершения выполнения метода. Свойство `Wait` воздействует только на следующий вызов какого-либо метода и после вызова этого метода снова принимает значение по умолчанию, поэтому при необходимости его следует переустановить. По умолчанию для методов `Play` и `StartRecording` свойство `Wait` имеет значение `False`, а для остальных методов — значение `True`.

Так, в процедуре

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  MediaPlayer1.Wait := True;
  MediaPlayer1.Play;
end;
```

компонент `MediaPlayer1` переводится в режим воспроизведения, при этом работа с приложением блокируется до окончания воспроизведения.

С нажатием кнопок мультимедийного проигрывателя связано несколько событий, из которых наиболее часто используются события `OnClick` и `OnNotify`.

Событие `OnClick` типа `EMPNotify` возникает при нажатии какой-либо кнопки. Тип `EMPNotify` описан так:

```
type EMPNotify = procedure(Sender: TObject; Button: TMPBtnType;
                           var DoDefault: Boolean) of object;
```

Параметр `Button` содержит информацию *о нажатой кнопке* и принимает следующие значения: `btPlay`, `btPause`, `btStop`, `btNext`, `btPrev`, `btStep`, `btBack`, `btRecord`, `btEject`. Перечисленные значения соответствуют приведенным ранее названиям кнопок проигрывателя, к которым добавлен префикс `bt`. Анализ этого параметра может понадобиться в случае, когда требуется определить, какую кнопку проигрывателя нажал пользователь.

Значение логического параметра `DoDefault` для нажатой кнопки определяет, *нужно ли выполнять стандартные действия*. Если параметр имеет значение `True` (по умолчанию), то для всех кнопок выполняются стандартные действия, например, при нажатии кнопки **Stop** вызывается соответствующий метод `Stop` компонента `MediaPlayer`. Если программист обрабатывает нажатия кнопок самостоятельно, то необходимый код размещается в обработчике события `OnClick`, а параметру `DoDefault` присваивается значение `False`.

Событие `OnNotify` типа `TNotifyEvent` возникает *при завершении* какого-либо из методов проигрывателя. Генерация этого события зависит от названия метода и значения свойства `Notify` типа `Boolean`. Событие `OnNotify` генерируется для каждого вызова методов `Play` и `StartRecording`, если перед их вызовом свойство `Notify` не было установлено в значение `False`. Другие методы проигрывателя, например, `Stop`, `Close` или `Next` не вызывают это событие, если свойство `Notify` не установлено в значение `True`. После обработки события `OnNotify` свойство `Notify` необходимо снова установить в значение `True` для обеспечения генерации следующего такого события.

Рассмотрим пример программы, реализующей проигрыватель мультимедийных файлов (рис. 11.4).

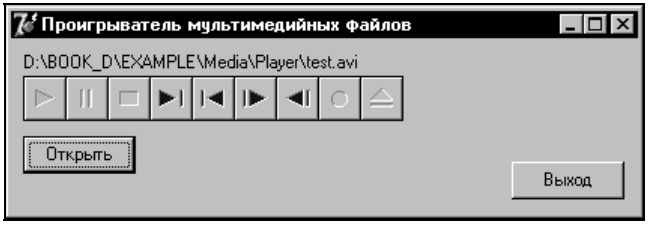


Рис. 11.4. Проигрыватель мультимедийных файлов

В листинге 11.3 приводится текст исходного модуля приложения с описанием формы и процедур обработки событий.

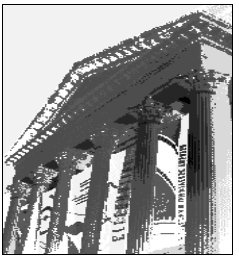
Листинг 11.3. Пример проигрывателя мультимедийных файлов

```
unit uMediaPl;
interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, MPlayer, StdCtrls, ExtCtrls, ComCtrls;
type
  TForm1 = class(TForm)
        btnClose: TButton;
    MediaPlayer1: TMediaPlayer;
     OpenDialog1: TOpenDialog;
         btnOpen: TButton;
          Label1: TLabel;

    procedure FormCreate(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure btnOpenClick(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  OpenDialog1.Filter :=
    'Файлы мультимедиа (*.wav; *.mid; *.avi) | *.wav;*.mid;*.avi';
  MediaPlayer1.DeviceType := dtAutoSelect;
  MediaPlayer1.AutoEnable := True;
  MediaPlayer1.AutoOpen   := False;
end;

procedure TForm1.btnOpenClick(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    Label1.Caption        := OpenDialog1.FileName;
    MediaPlayer1.FileName := OpenDialog1.FileName;
    MediaPlayer1.Open;
  end;
end;

procedure TForm1.btnCloseClick(Sender: TObject);
begin
  Close;
end;
end.
```

Выбор файла для воспроизведения осуществляется в диалоговом окне открытия файла, которое появляется при нажатии кнопки **Открыть** (`btnOpen`). В случае подтверждения выбора указанный файл загружается в проигрыватель, а его имя отображается в надписи `Label1`. Пользователь управляет проигрывателем с помощью кнопок.

ГЛАВА 12



Работа с файлами и каталогами

При организации многих приложений требуется работа с файлами и каталогами (папками). В модулях `System` и `SysUtils` находятся многочисленные процедуры, функции, типы и константы, предназначенные для работы с файловой системой. При создании новой формы названия обоих модулей автоматически вносятся в раздел `uses` модуля формы. Если же подготовлен модуль, не связанный непосредственно с формой, то раздел `uses` следует дополнить самостоятельно.

Для работы с файловой системой предназначены четыре специальных компонента (`FileListBox`, `DirectoryListBox`, `DriveComboBox`, `FilterComboBox`), расположенные на вкладке **Win3.1** Палитры компонентов. Однако, несмотря на название вкладки, использующие эти компоненты программы корректно работают и в 32-разрядных версиях Windows.

Работу с файлами соответствующих форматов поддерживают также отдельные компоненты и объекты, например, `Memo`, `ListBox`, `Picture` и `Clipboard`.

Работа с инициализационными файлами и системным реестром рассматривается в *главе 36*.

Средства системных модулей

Для файловых операций часто используются средства системных модулей `System` и `SysUtils`.

Модуль `System` реализует *средства низкого уровня*, в том числе средства ввода/вывода, обработки строк, операций с числами и динамической памятью. Многие из этих средств используются компилятором Delphi.

Модуль `SysUtils` *расширяет* и *дополняет возможности* модуля `System`. В нем находятся средства, предназначенные для работы с дисками, каталогами и файлами, датой, временем, строками, определяются классы исключений, а также содержатся другие полезные процедуры, функции и константы.

Отдельные возможности модуля `SysUtils` базируются на соответствующих возможностях модуля `System` или дублируют их.

Для операций с атрибутами (табл. 12.1) и режимами открытия (табл. 12.2) файлов можно использовать именованные константы модуля SysUtils.

Таблица 12.1. *Константы атрибутов файла*

Константа	Значение	Назначение
faReadOnly	$000000001	Файл только для чтения
faHidden	$000000002	Скрытый файл
faSysFile	$000000004	Системный файл
faVolumeID	$000000008	Метка диска
faDirectory	$000000010	Каталог
faArchive	$000000020	Архивный файл
faSymLink	$000000040	Файл — символическая связь
faAnyFile	$000000047	Любой файл

Каждая константа содержит единицу в соответствующем бите шестнадцатеричного числа. Для получения требуемого значения файловых атрибутов приведенные константы можно объединять с помощью поразрядной операции OR. Например, параметр для работы с системными и скрытыми файлами получается так:

```
faHidden or faSysFile
```

В константе faAnyFile биты всех атрибутов установлены в единицу. Значение константы faAnyFile можно получить также следующим образом:

```
faAnyFile = faArchive or faDirectory or faVolumeID or
            faSysFile or faHidden or faReadOnly;
```

Константы режимов открытия файла служат для задания уровня доступа к открытому файлу, а также к потоку, например, в функции FileOpen.

Таблица 12.2. *Константы режимов открытия файла*

Константа	Значение	Режим открытия файла
fmOpenRead	$0000	Только для чтения
fmOpenWrite	$0001	Только для записи
fmOpenReadWrite	$0002	Для чтения и записи
fmShareCompat	$0004	Совместим с файлом, открытым по FCB
fmShareExclusive	$0010	Открыт для монопольного использования
fmShareDenyWrite	$0020	Закрыт для записи другим пользователем
fmShareDenyRead	$0030	Закрыт для чтения другим пользователем
fmShareDenyNone	$0040	Открыт для совместного использования
fmClosed	$D7B0	Закрыт
fmInput	$D7B1	Открыт для ввода

Таблица 12.2 *(окончание)*

Константа	Значение	Режим открытия файла
fmOutput	$D7B2	Открыт для вывода
fmInOut	$D7B3	Открыт для ввода/вывода

Для удобства работы с именами файлов в модуле SysUtils вводится тип TFileName, который равен типу String.

Содержащиеся в модуле SysUtils подпрограммы позволяют выполнять множество разнообразных операций с дисками, каталогами и файлами.

Для определения *размера диска* и *свободного пространства* на диске предназначены функции DiskSize(Drive: Byte): Int64 и DiskFree(Drive: Byte): Int64. В качестве результата обе функции возвращают соответствующий размер диска (полный или свободный) в байтах, в случае ошибки возвращается значение -1. Номер дискового устройства задается целым числом в параметре Drive:

- 0 (текущий диск);
- 1 (диск A:);
- 2 (диск B:);
- 3 (диск C:)

и т. д.

Если задан номер несуществующего устройства, то исключение не генерируется, а в качестве результата возвращается нулевое значение.

Так, в процедуре

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := 'Объем диска D: — ' + IntToStr(DiskSize(4)
                    div 1024) + ' Килобайт';
  Label2.Caption := 'На диске D: свободно — ' + IntToStr(DiskFree(4)
                    div 1024) + ' Килобайт';
end;
```

в надписях Label1 и Label2 выводится информация о полном и свободном объемах дискового пространства на диске D: в килобайтах.

Ряд подпрограмм модуля SysUtils предназначен для работы с *каталогами*.

Функция GetCurrentDir: String возвращает строку, содержащую значение *текущего каталога* Windows. Для *смены каталога* можно использовать функцию SetCurrentDir(const Dir: string): Boolean. Функции CreateDir(const Dir: string): Boolean и RemoveDir(const Dir: string): Boolean позволяют *создать* новый и *удалить* существующий каталог соответственно. Имя каталога задается параметром Dir. Удалить можно только пустой каталог. В случае успешного выполнения операции функции возвращают значение True и значение False — в противном случае.

Аналогичное назначение имеют процедуры GetDir, ChDir, MkDir и RmDir модуля System.

Для *смены* текущего каталога, кроме функции `SetCurrentDir`, можно также использовать диалоговое окно выбора каталога, вызываемое функцией `SelectDirectory`. Эта функция входит в состав модуля `FileCtrl`, и в случае ее использования следует подключить данный модуль в разделе `uses`.

При вызове функции `SelectDirectory(var Directory: string; Options: TSelectDirOpts; HelpCtx: Longint): Boolean` параметр `Directory` задает каталог, который первоначально предлагается пользователю для выбора. В случае успешного выбора каталога и закрытия диалогового окна кнопкой **OK** параметр `Directory` будет содержать имя каталога, а функция возвратит значение `True`. Параметр `Options` определяет параметры диалогового окна и может содержать набор перечисленных ниже значений.

- `sdAllowCreate` (пользователь может в поле редактирования ввести несуществующий каталог). Создается при этом новый каталог или нет, зависит от значения `sdPerformCreate`.
- `sdPerformCreate` (применяется совместно с параметром `sdAllowCreate`). Если каталог, введенный пользователем, не существует, то он создается. Если этот параметр отключен, то возможно создание введенного каталога после закрытия диалога `SelectDirectory`.
- `sdPrompt` (применяется совместно с параметром `sdAllowCreate`). Служит для выдачи предупреждения о вводе пользователем несуществующего каталога и запроса на его создание. Создание каталога зависит от параметра `sdPerformCreate`.

Так, в процедуре:

```
// В разделе uses следует подключить модуль FileCtrl
procedure TForm1.Button2Click(Sender: TObject);
var Path: string;
begin
  Path := GetCurrentDir;
  Label3.Caption := 'Текущий каталог: ' + Path;
  if SelectDirectory(Path, [sdAllowCreate,sdPerformCreate,sdPrompt], 0)
    then Label3.Caption := 'Текущий каталог: ' + Path;
end;
```

пользователю предлагается выбрать новый текущий каталог. Название каталога отображается в надписи `Label3`. Пользователь может создать новый каталог.

Для операций с файлами модуль `SysUtils` содержит следующие функции:

- `FileCreate (const FileName: String): Integer` (создает файл, в качестве результата возвращает дескриптор файла);
- `RenameFile (const OldName, NewName: String): Boolean` (переименовывает файл);
- `DeleteFile (const FileName: String): Boolean` (удаляет файл);
- `FileGetAttr (const FileName: String): Integer` (возвращает атрибуты файла);
- `FileSetAttr (const FileName: String; Attr: Integer): Integer` (устанавливает атрибуты файла).

В функциях модуля `SysUtils` параметры `FileName` и `OldName` указывают полные имена файлов.

Для получения и установки параметров файла (имя, дата и время создания, атрибуты и др.) используется специальная переменная типа `TSearchRec`. Ее описание:

```
TSearchRec = record
          Time: Integer;
          Size: Integer;
          Attr: Integer;
          Name: TFileName;
   ExcludeAttr: Integer;
    FindHandle: THandle;
      FindData: TWin32FindData;
end;
```

Рассмотренные функции, возвращающие целочисленные значения, в случае ошибки возвращают значение `-1`.

Все логические функции модуля `SysUtils` в случае нормального завершения возвращают значение `True`, а в случае ошибки — `False`.

Для выполнения рассмотренных операций с файлами также можно использовать процедуры модуля `System`, такие как `Rename`, `Erase`, `FileOpen`, `FileClose` и др.

В следующей инструкции переименовывается файл test3.doc:

```
if not RenameFile('test3.doc', Edit1.Text)
  then MessageDlg('Файл не переименован!'. MtError, [mbOK], 0);
```

Новое имя файла вводится в элементе редактирования `Edit1`. В случае ошибки при выполнении операции переименования выдается предупреждающее сообщение.

Модуль `SysUtils` содержит следующие подпрограммы, предназначенные для *поиска файлов* и *проверки наличия файлов* с заданными атрибутами:

- `FileExists(const FileName: String): Boolean` (проверяет наличие файла);
- `FileSearch(const Name, DirList: String): String` (выполняет поиск файла в каталогах, заданных параметром `DirList`); в случае успешного поиска возвращается полное имя файла, и пустая строка — в противном случае;
- `FindFirst(const Path: string; Attr: Integer; var F: TSearchRec): Integer` (выполняет поиск файла в каталоге `Path`); параметр `Attr` содержит атрибуты файла (можно задавать с помощью констант), результат поиска размещается в параметре `TSearchRec`;
- `FindNext(var F: TSearchRec): Integer` (выполняет поиск следующего файла с параметрами, заданными функцией `FindFirst`);
- `FindClose(var F: TSearchRec)` (завершает процесс поиска, выполняемого с помощью функций `FindFirst` и `FindNext`).

Например, в инструкции

```
if FileExists (Edit1.Text) then DeleteFile(Edit1.Text);
```

перед удалением файла, имя которого набрано в редакторе `Edit1.Text`, проверяется его существование, а в инструкции

```
Label1.Caption := FileSearch('Example.pas', 'D:\;' + 'D:\BOOK\');
```

выполняется поиск файла Example.pas в каталогах D: и D:\BOOK. Каталоги в списке разделяются точкой с запятой (;). Полное имя найденного файла отображается в надписи Label1.

Подпрограммы FindFirst, FindNext и FindClose используются совместно. Функция FindFirst *начинает процесс поиска* файлов, в ней задаются путь для поиска и атрибуты искомых файлов. Функция FindNext начинает *поиск следующего файла*, удовлетворяющего ранее заданным условиям. В случае успешного поиска обе функции возвращают в качестве результата нулевое значение и другое число — в противном случае. Процедура FindClose *завершает процесс поиска*, начатый функцией FindNext.

Пример поиска файлов:

```
procedure TForm1.Button3Click(Sender: TObject);
label 10;
var SR: TSearchRec;
begin
  if FindFirst('D:\WORK\*.*', faArchive, SR) = 0 then begin
     Label1.Caption := SR.Name + ' — ' + IntToStr(SR.Size);
10: if MessageDlg('Искать далее?', mtConfirmation, [mbYes, mbNo], 0) = mrYes
      then
        if FindNext(SR) = 0 then begin
          Label1.Caption := SR.Name+' — ' + IntToStr(SR.Size);
          goto 10;
        end;
    end;
  FindClose(SR);
  MessageDlg('Поиск прекращен.', mtConfirmation, [mbOK], 0);
end;
```

В приведенной процедуре выполняется последовательный просмотр архивных файлов каталога D:\WORK. В случае успешного поиска в надписи Label1 выводится название найденного файла и его размер в байтах.

В составе модуля SysUtils для работы с *именами дисков*, *каталогов* и *файлов* предназначены следующие функции:

- ExtractFileDrive(const FileName: String): String (возвращает имя дискового устройства, на котором находится файл);
- ExtractFilePath(const FileName: String): String (возвращает путь к файлу, путь содержит имя дисковода и конечный обратный слэш (\));
- ExtractFileDir(const FileName: String): String (возвращает путь к файлу без конечного обратного слэша (\), путь содержит имя дисковода);
- ExtractFileName(const FileName: String): String (возвращает имя файла без пути, у имени есть расширение и разделительная точка);
- ExtractFileExt(const FileName: String): String (возвращает расширение имени файла с разделительной точкой);
- ChangeFileExt(const FileName, Extension: String): String (изменяет расширение файла, заданное параметром Extension; расширение состоит из четырех символов,

первым из которых является точка); функция возвращает в качестве результата имя файла с новым расширением, само имя файла не изменяется;

- `ExpandFileName(const FileName: String): String` (возвращает полное имя файла, включая дисковое устройство и путь).

В приводимой далее процедуре надписи отображают диск, путь и имя для исполняемого файла приложения:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := ExtractFileDrive(Application.ExeName);
  Label2.Caption := ExtractFileDir(Application.ExeName);
  Label3.Caption := ExtractFileName(Application.ExeName);
end;
```

В состав модуля `System` входят следующие подпрограммы, предназначенные для выполнения операций, связанных с записью в файл:

- `AssignFile(var F; FileName: String)` (устанавливает связь между логическим файлом (файловой переменной) и физическим файлом, заданным параметром `FileName`);
- `Reset(var F [:File; RecSize: Word])` (открывает файл для чтения);
- `Rewrite(var F [:File; RecSize: Word])` (открывает файл для записи);
- `Append(var F: Text)` (открывает файл для добавления информации в его конец); только для текстовых файлов;
- `CloseFile(var F)` (закрывает файл);
- `Read(F, P1,...,Pn)` (считывает из файла значения указанных переменных);
- `ReadLn(F, P1,...,Pn)` (считывает из файла значения указанных переменных с переходом на новую строку); только для текстовых файлов;
- `Write(F, V1,...,Vn)` (записывает в файл значения указанных выражений);
- `Writeln(F, V1,...,Vn)` (записывает в файл значения указанных выражений и переходит на новую строку); только для текстовых файлов;
- `FileSize(var F): Integer` (возвращает размер файла в виде числа содержащихся в файле записей); только для нетекстовых файлов;
- `Eof(var F): Boolean` (возвращает значение `True`, если достигнут конец файла, и `False` — в противном случае);
- `Seek(var F; N: Longint)` (перемещает текущий указатель на число записей, заданное параметром `N`, перемещение выполняется относительно начала файла); для текстовых файлов необходимо использовать функцию `SeekEof`.

В названных подпрограммах параметр `F` указывает имя логического файла (файловой переменной), с которым выполняются операции. В квадратных скобках указаны необязательные элементы.

Приведенные выше процедуры аналогичны соответствующим процедурам `Assign`, `Reset`, `Rewrite`, `Append`, `Close`, `Read`, `ReadLn`, `Write` и `Writeln` языка Pascal. Отличие заключается в названиях процедур `AssignFile` и `CloseFile`: поскольку в Delphi методы `Assign`

и Close используются для других целей, в названия методов модуля System включено слово File.

Для использования средств модуля System в работе с текстовыми файлами при описании файловой переменной типа Text нужно явно указать модуль System:

```
var f: System.Text;
```

Это обусловлено тем, что слово Text используется Delphi по другому назначению. Кроме того, для описания файловой переменной текстового типа можно использовать тип TextFile, специально введенный для этого.

Рассмотрим следующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
var f: System.Text;
begin
  AssignFile(f, 'D:\List.txt');
  Rewrite(f);
  for n := 0 to Memo1.Lines.Count — 1 do writeln(f, Memo1.Lines[n]);
    FileClose(f);
end;

procedure TForm1.Button2Click(Sender: TObject);
var f2: System.Text;
begin
  AssignFile(f2, 'D:\List.txt');
  Reset(f2);
  while not eof (f2) do begin
      readln(f2, s);
      ListBox1.Add(s);
  end;
  FileClose(f2);
end;
```

При нажатии кнопки Button1 содержимое редактора Memo1 записывается в текстовый файл D:\List.txt. Если указанный файл не существует, то он создается; если же файл существует, то он перезаписывается. При нажатии кнопки Button2 содержимое файла D:\List.txt заносится в список ListBox1. Ошибки открытия, записи и чтения из файла не проверяются. Для работы с текстовым файлом в обработчиках нажатия кнопок разными способами описаны файловые переменные f и f2. Отметим, что данный пример выглядел бы проще при использовании методов WriteToFile и ReadFromFile класса TStrings.

При операциях с файлами Delphi обрабатывает исключения с помощью глобального обработчика. При желании программист может добавить и собственную обработку исключений. Кроме того, можно обрабатывать ошибки ввода/вывода способом, традиционным для языка Pascal: встроенная обработка ошибок отключается путем установки в нужном месте программы директивы {$I-} (по умолчанию она включена, т. е. установлена директива {$I+}), после чего анализ ошибок выполняется с помощью функции IOResult: Integer, возвращающей код ошибки операции ввода/вывода. При необходимости встроенную обработку ошибок ввода/вывода можно снова включить, указав директиву {$I+}.

Следующие подпрограммы модуля `SysUtils` предназначены для операций чтения из файлов и записи в файлы.

- `FileOpen(const FileName: String; Mode: Integer): Integer` (открывает файл). Параметр `Mode` задает режим доступа. В качестве результата возвращается дескриптор открытого файла.
- `FileClose(FileHandle: Integer)` (закрывает файл).
- `FileRead(FileHandle: Integer; var Buffer; Count: Integer): Integer` (считывает из открытого файла в буфер, указанный параметром `Buffer`, столько байтов данных, сколько задано параметром `Count`).
- `FileWrite(FileHandle: Integer; var Buffer; Count: Integer): Integer` (записывает в открытый файл из буфера, указанного параметром `Buffer`, столько байтов данных, сколько задано параметром `Count`).
- `FileSeek(Handle, Offset, Origin: Integer): Integer` (служит для перемещения по файлу). Параметр `Offset` задает в байтах смещение, куда необходимо позиционировать указатель. Начало отсчета смещения указывает параметр `Origin`, принимающий следующие значения:
  - `0` (от начала файла);
  - `1` (от текущей позиции указателя);
  - `2` (от конца файла).

В названных подпрограммах параметр `FileHandle` содержит дескриптор файла.

Внимание!

Процедуры `CloseFile` и `FileClose` модулей `System` и `SysUtils` имеют похожие имена, что может привести к ошибке при их вызове.

Компоненты для работы с файлами и каталогами

Для работы с файлами и каталогами в Delphi предназначены следующие четыре специальных компонента (рис. 12.1):

- `FileListBox` (просмотр списка и выбор имени файла);
- `DirectoryListBox` (просмотр и перемещение по дереву каталогов);
- `DriveComboBox` (выбор дисковода);
- `FilterComboBox` (выбор маски для списка файлов).

С помощью этих компонентов можно программировать операции, связанные с навигацией по файловой системе, например, организовывать диалоги выбора имен открываемых или сохраняемых файлов. Обычно компоненты, связанные с файлами и каталогами, используются совместно, но при необходимости программист может применять их по отдельности. На рис. 12.2 показаны связи компонентов при их совместном использовании (в данном случае для организации диалога выбора файла для открытия).

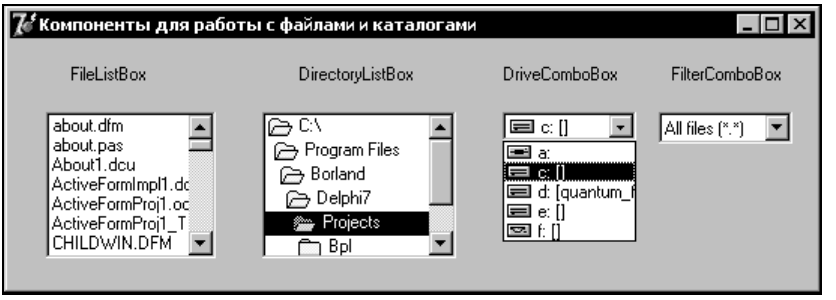


Рис. 12.1. Компоненты для операций с файлами и каталогами

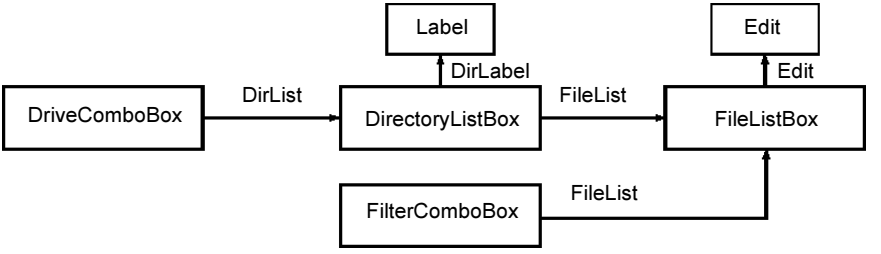


Рис. 12.2. Связи между компонентами

Кроме четырех рассмотренных компонентов, на рис. 12.2 показаны также элементы `Label` и `Edit`. Над соединительными линиями указаны свойства, с помощью которых устанавливается связь между компонентами (свойство принадлежит элементу, из которого линия выходит).

В следующих разделах подробно рассмотрены все четыре специальных компонента и приведен пример их использования в приложении.

Компонент *DriveComboBox*

Для *выбора дисковода* используется компонент `DriveComboBox`. Этот компонент представляет собой комбинированный список, содержащий все установленные в системе дисковые накопители. Список раскрывается, и пользователь может выбрать в нем нужное устройство. Выбранное устройство отображается в поле компонента, а соответствующая выбранному дисководу буква содержится в свойстве `Drive` типа `Char`. Значение свойства `Drive` также можно устанавливать программно, например:

```
DriveComboBox1.Drive := 'D';
```

Замечание

Если указанное устройство в системе отсутствует, то инструкция присваивания игнорируется.

Аналогичное свойство `Drive` также имеется у компонентов `DirectoryListBox` и `FileListBox`. Если компоненты `DriveComboBox`, `DirectoryListBox` и `FileListBox` используются совместно, то после смены в элементе `DriveComboBox` устройства нужно соответ-

ственно изменить значение свойства Drive для двух других компонентов. Обычно это выполняется в обработчике события OnChange, возникающего при выборе нового дисковода.

Например, в процедуре

```
procedure TForm1.DriveComboBox1Change(Sender: TObject);
begin
  // Удобнее организовать эту связь через свойство DirList
  DirectoryListBox1.Drive := DriveComboBox1.Drive;
  // Удобнее организовать эту связь через свойство FileList
  FileListBox1.Drive := DriveComboBox1.Drive;
end;
```

при выборе нового устройства обновляются значения свойства Drive компонентов DirectoryListBox1 и FileListBox1. Для компонента FileListBox1, кроме того, устанавливается новый каталог.

Более удобным способом связи компонента DriveComboBox с компонентом DirectoryListBox является использование свойства DirList, т. к. в этом случае при смене дисковода обновление значений свойств Drive обоих компонентов происходит автоматически.

Список всех дисковых устройств при выполнении программы доступен через свойство Items типа TStrings. В случае необходимости можно присвоить этот список другому объекту совместимого с TStrings типа.

Пример получения списка дисковых устройств:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items.Assign(DriveComboBox1.Items);
end;
```

Свойство TextCase типа TTextCase определяет, в каком регистре задаются буквы дисковых устройств. Это свойство принимает следующие значения:

- tcLowerCase (буквы устройств задаются в нижнем регистре) — по умолчанию;
- tcUpperCase (буквы устройств задаются в верхнем регистре).

При навигации по файловой системе компонент DriveComboBox часто связывается через свойство DirList типа TDirectoryListBox с компонентом DirectoryListBox. В этом случае при смене дисковода элемент DirectoryListBox автоматически формирует и отображает новое дерево каталогов. Если компонент DirectoryListBox в свою очередь через свойство FileList связан с компонентом FileListBox, то список файлов также обновляется соответствующим образом.

Компонент *DirectoryListBox*

Для *просмотра* и *перемещения по дереву каталогов* служит компонент DirectoryListBox, представляющий собой прямоугольную область, в которой отображается дерево каталогов с выбранным текущим каталогом. Пользователь может перемещаться по этому дереву.

Свойство `Directory` типа `String` содержит строку, указывающую *текущий каталог*. Каталог может быть выбран пользователем или установлен программно, например, через инструкцию присваивания:

```
DirectoryListBox1.Directory := 'C:\GAMES';
```

Замечание

Если указанный каталог отсутствует, то генерируется исключение.

Для *отображения текущего каталога* с компонентом `DirectoryListBox` через свойство `DirLabel` типа `TLabel` может быть связана надпись `Label`. Эта надпись обычно располагается над компонентом `DirectoryListBox` и автоматически отображает каталог в свойстве `Caption`. При этом на длину строки, содержащейся в свойстве `Caption` надписи, накладывается ограничение — не более 24 символов. Если название текущего каталога имеет длину больше, чем 24 символа (такое бывает при большой вложенности каталогов), то часть названий каталогов заменяется многоточием. Например, каталог

```
C:\Program Files\Borland\Delphi 7\Bin
```

в надписи, связанной с компонентом `DirectoryListBox` через свойство `DirLabel`, будет отображен так:

```
C:\...\Borland\Delphi 7\Bin
```

При необходимости *полного отображения* названий каталогов можно использовать надпись `Label`, не связанную с компонентом `DirectoryListBox`, значение которой присваивается в обработчике события `OnChange`, возникающем при смене каталога. Другие компоненты, предназначенные для операций с файлами и каталогами, также не имеют свойства `Caption`, поэтому обычно рядом с ними, как правило, располагают надпись, поясняющую назначение элемента управления.

Компонент `DirectoryListBox` обычно связывается через свойство `FileList` типа `TFileListBox` с компонентом `FileListBox`. В этом случае при смене каталога компонент `FileListBox` автоматически сформирует и отобразит новый список файлов, соответствующий выбранному каталогу.

Компонент *FileListBox*

Для *просмотра списка* файлов заданного каталога и *выбора имени* файла предназначен компонент `FileListBox`. Он представляет собой прямоугольную область, в которой отображается список файлов заданного каталога с возможностью визуального выбора файлов.

Просматриваемый *каталог* задается свойством `Directory` типа `String`. Чаще всего компоненты `FileListBox` и `DirectoryListBox` используются совместно и связаны через свойство `FileList`, поэтому при смене каталога список файлов обновляется автоматически. Если компонент `FileListBox` применяется отдельно, то значение свойства `Directory` должен устанавливать программист.

При выполнении программы список файлов каталога доступен через свойство `Items` типа `TStrings`. К списку применимы операции, допустимые для класса `TStrings`.

Для автоматического *отображения* выбранного в списке *файла* можно использовать компонент `TEdit`, который должен быть связан с компонентом `FileListBox` с помощью указывающего на `TEdit` свойства `FileEdit` типа `TEdit`. Первоначально, когда выбранных файлов нет, в поле `Edit` отображается выбранная маска, например, `*.*`.

Если список поддерживает одновременный *выбор нескольких файлов* (свойство `MultiSelect` имеет значение `True`), то вместо однострочного редактора `Edit` для отображения выбранных файлов можно использовать компонент-список `ListBox` или многострочный редактор `Memo`. В этом случае выбранные файлы не отображаются автоматически, и программист должен кодировать соответствующие операции самостоятельно.

При выборе пользователем файла в свойство `FileName` типа `String` заносится *полное имя* выбранного файла. Это свойство доступно для чтения и записи, поэтому можно программно выбрать требуемый файл, установив в качестве значения свойства `FileName` имя этого файла. Например:

```
FileListBox1.FileName := 'filecom.exe';
```

Замечание

Если указанный файл отсутствует в каталоге, то генерируется исключение.

Управлять отображением файлов в списке можно с помощью свойств `FileType` и `Mask`. Свойство `FileType` типа `TFileType` указывает *атрибуты* файлов, содержащихся в списке, и принимает комбинации следующих значений:

- `ftReadOnly` (только чтение);
- `ftHidden` (скрытый);
- `ftSystem` (системный);
- `ftVolumeID` (метка диска);
- `ftDirectory` (каталог);
- `ftArchive` (архивный);
- `ftNormal` (обычный).

Эти значения соответствуют константам атрибутов файлов, определенным в модуле `SysUtils`. По умолчанию свойство `FileType` имеет значение `[ftNormal]`, и отображаются все файлы, не имеющие специальных атрибутов.

Свойство `Mask` типа `String` содержит *маску* для файлов списка. По умолчанию используется маска `*.*`, и в списке отображаются файлы с любыми именами и типами. При необходимости для маски можно задать другое значение, например, `*.bmp;*.pcx` для отображения списка файлов, содержащих растровую графику (формирование маски было рассмотрено в *главе 4* при описании стандартных диалоговых окон, связанных с выбором имени файла для открытия или сохранения). При выполнении приложения маску для компонента `FileListBox` можно брать из одноименного свойства компонента `FilterComboBox`. Удобнее осуществлять фильтрацию файлов с помощью связи компонентов `FileListBox` и `FilterComboBox` через свойство `FileList` последнего *(см. следующий раздел)*.

Файлы в списке могут выводиться со значками или без них, наличие или отсутствие *значка* задается свойством `ShowGlyphs` типа `Boolean`. По умолчанию это свойство имеет значение `False`, и у файлов значки отсутствуют. Если установить свойство `ShowGlyphs`

в значение `True`, то слева от имен файлов выводятся значки. Отображение значков повышает наглядность списка, но несколько замедляет процесс отображения.

Компонент *FilterComboBox*

Выбор маски для списка файлов удобно осуществлять с помощью компонента `FilterComboBox`. Этот компонент представляет собой комбинированный список, который содержит фильтры. Пользователь может выбрать требуемый фильтр в предложенном списке, выбранный фильтр отображается в текстовом поле компонента `FilterComboBox`.

Напомним, что *фильтр* состоит из текстового описания и маски. Описание поясняет для пользователя назначение маски, а маска служит для отбора группы файлов с определенными именами и типами. Свойства `Filter` и `Mask` типа `String` компонента `FilterComboBox` позволяют получить доступ и управлять фильтром и маской соответственно. Отдельные элементы фильтра доступны через свойство `Items` типа `TStrings`.

При совместном использовании компонентов `FilterComboBox` и `FileListBox` для организации связи между ними используется свойство `FileList` типа `TFileListBox`. После установки связи между этими компонентами в случае смены в компоненте `FilterComboBox` фильтра, а вместе с ним и маски, список файлов в компоненте `FileListBox` автоматически обновляется с учетом новой маски.

Пример приложения

В качестве примера использования компонентов, предназначенных для работы с файлами и каталогами, рассмотрим приложение, реализующее просмотр графических файлов: растровой графики (bmp, pcx), метафайлов (wmf) и значков (ico). Приложение включает в свой состав две формы: главную и диалоговую.

В главной форме (рис. 12.3) расположены: компонент `Image1`, в котором отображается выбранный графический файл, надпись `Label1`, содержащая имя файла, и две кнопки — `btnOpen` (открывающая диалоговое окно выбора имени файла) и `btnExit` (завершает приложение).

Рис. 12.3. Главная форма приложения, реализующего просмотр графических файлов

По умолчанию свойство `Stretch` компонента `Image1` имеет значение `False`, т. е. если рисунок не умещается целиком в отведенной под компонент области, то часть рисунка отсекается. Для полного просмотра больших изображений, находящихся в компоненте `Image1`, нужно установить свойству `Stretch` значение `True` (другим вариантом является

прокрутка изображения, *см. главу 10*, посвященную графическим возможностям Delphi).

В листинге 12.1 приводится код модуля `uMain` главной формы `fmMain`.

Листинг 12.1. Код модуля главной формы приложения, реализующего просмотр графических файлов

```
unit uMain;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TfmMain = class(TForm)
     Image1: TImage;
    btnOpen: TButton;
    btnExit: TButton;
     Label1: TLabel;
    procedure btnOpenClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var  fmMain: TfmMain;
implementation
uses uOpen;
{$R *.DFM}

procedure TfmMain.FormCreate(Sender: TObject);
begin
  Label1.Caption := '';
end;

procedure TfmMain.btnOpenClick(Sender: TObject);
begin
  if fmOpen.ShowModal = mrOK then begin
     Label1.Caption := fmOpen.FileListBox1.FileName;
     Image1.Picture.LoadFromFile(fmOpen.FileListBox1.FileName);
  end;
end;

procedure TfmMain.btnExitClick(Sender: TObject);
begin
  Close;
end;
```

При нажатии кнопки `btnOpen` вызывается модальная диалоговая форма `fmOpen`, в которой пользователь выбирает имя графического файла. В случае выбора имени файла и

закрытия диалогового окна нажатием кнопки **OK** указанный файл загружается в компонент `Image1`, а имя файла выводится в надписи `Label1`. Имя файла берется из свойства `FileName` компонента `FileListBox1`, находящегося в диалоговой форме (для хранения имени выбранного в диалоговом окне файла можно также объявить и использовать далее специальную переменную).

В диалоговой форме (рис. 12.4) находятся компоненты `DriveComboBox1`, `DirectoryListBox1`, `FileListBox1`, `FilterComboBox1`, `lblPath` и `edtFile`. Организация связи между ними выполнена при создании формы, а не через Инспектор объектов. В правой части окна расположены компонент `Image1`, служащий для предварительного просмотра изображения, а также кнопки закрытия диалогового окна `btnOK` и `btnCancel`.

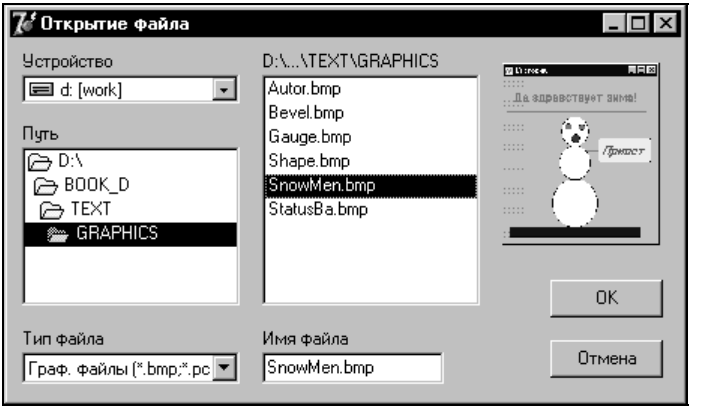


Рис. 12.4. Диалоговая форма приложения для просмотра графических файлов

Свойство `Filter` компонента `FilterComboBox1` указывает на список масок для графических файлов, которые позволяет просматривать данное приложение. Для заполнения фильтра использована функция `GraphicFilter`.

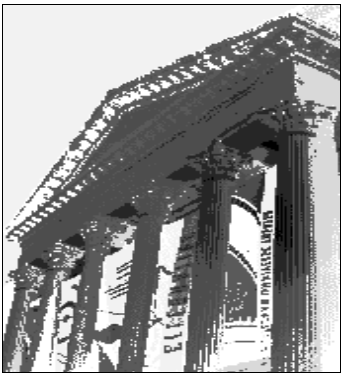
В листинге 12.2 приводится код модуля `uOpen` диалоговой формы `fmOpen` выбора имени файла.

Листинг 12.2. Код модуля диалоговой формы выбора имени файла

```
unit uOpen;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, FileCtrl, StdCtrls, ExtCtrls;
type
  TfmOpen = class(TForm)
        FileListBox1: TFileListBox;
    DirectoryListBox1: TDirectoryListBox;
       DriveComboBox1: TDriveComboBox;
      FilterComboBox1: TFilterComboBox;
               Label1: TLabel;
              lblPath: TLabel;
               Label3: TLabel;
```

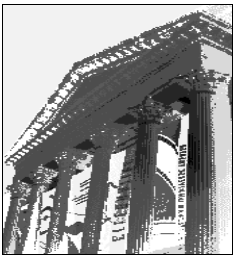
```
    Label4: TLabel;
     btnOK: TButton;
   edtFile: TEdit;
    Label2: TLabel;
    Image1: TImage;
 btnCancel: TButton;

  procedure FormCreate(Sender: TObject);
  procedure FileListBox1Change(Sender: TObject);

 private
  { Private declarations }
 public
  { Public declarations }
 end;
var fmOpen: TfmOpen;
implementation
{$R *.DFM}

procedure TfmOpen.FormCreate(Sender: TObject);
begin
 DriveComboBox1.DirList     := DirectoryListBox1;
 DirectoryListBox1.FileList := FileListBox1;
 DirectoryListBox1.DirLabel := lblPath;
 FileListBox1.FileEdit      := edtFile;
 FilterComboBox1.FileList   := FileListBox1;
 FilterComboBox1.Filter     := GraphicFilter(TGraphic);
 Image1.Stretch             := True;
 btnOK.ModalResult          := mrOK;
 btnCancel.ModalResult      := mrCancel;
end;

procedure TfmOpen.FileListBox1Change(Sender: TObject);
begin
 if FileListBox1.FileName = '' then begin
    Image1.Picture := Nil;
    btnOK.Enabled  := False;
 end
 else begin
    Image1.Picture.LoadFromFile(FileListBox1.FileName);
    btnOK.Enabled := True;
 end;
end;
end.
```

Выбранный графический файл загружается в компонент `Image1` для предварительного просмотра в обработчике события `OnChange` списка файлов `FileListBox1`. Если выбранного файла нет, то содержимое компонента `Image1` очищается, а кнопка `btnOK` закрытия диалогового окна блокируется. Чтобы просматриваемый рисунок полностью отображался в компоненте `Image1`, при создании окна свойство `Stretch` этого компонента установлено в значение `True`.



ЧАСТЬ III

Основы работы с базами данных



ГЛАВА 13

Основные понятия баз данных

Часто для успешного функционирования различным организациям требуется развитая информационная система, реализующая автоматизированный процесс сбора, манипулирования и обработки данных.

Банки данных

Современной формой информационных систем являются *банки данных*, имеющие в своем составе:

- вычислительную систему;
- систему управления базами данных (СУБД);
- одну или несколько баз данных (БД);
- набор прикладных программ (приложений БД).

База данных (БД) обеспечивает хранение информации, а также удобный и быстрый доступ к данным. Она представляет собой совокупность данных различного характера, организованных по определенным правилам. Информация в БД должна быть:

- непротиворечивой;
- неизбыточной;
- целостной.

Система управления базой данных (СУБД) — это совокупность языковых и программных средств, предназначенных для создания, ведения и использования БД. По характеру применения СУБД разделяют на персональные и многопользовательские.

Персональные СУБД обеспечивают возможность создания локальных БД, работающих на одном компьютере. К персональным СУБД относятся Paradox, dBase, FoxPro, Access и др.

Замечание

СУБД Access 97/2000/2002 также обеспечивают возможность многопользовательского доступа к данным.

Многопользовательские СУБД позволяют создавать информационные системы, функционирующие в архитектуре "клиент-сервер". Наиболее известными многопользовательскими СУБД являются Oracle, Informix, SyBase, Microsoft SQL Server, InterBase.

В состав языковых средств современных СУБД входят:

- язык описания данных, предназначенный для описания логической структуры данных;
- язык манипулирования данными, обеспечивающий выполнение основных операций над данными — ввод, модификацию и выборку;
- язык структурированных запросов (SQL, Structured Query Language), обеспечивающий управление структурой БД и манипулирование данными, а также являющийся стандартным средством доступа к удаленным БД;
- язык запросов по образцу (QBE, Query By Example), обеспечивающий визуальное конструирование запросов к БД.

Прикладные программы, или *приложения*, служат для обработки данных, содержащихся в БД. Пользователь осуществляет управление БД и работу с ее данными именно с помощью приложений, которые также называют *приложениями БД*.

Иногда термин "база данных" трактуют в более широком смысле и обозначают им не только саму БД, но и приложения, обрабатывающие ее данные.

Замечание

Хотя система Delphi и не является СУБД в буквальном смысле этого слова, тем не менее, она обладает вполне развитыми возможностями СУБД. Предоставляемые Delphi средства обеспечивают создание и ведение локальных и клиент-серверных БД, а также разработку приложений для работы практически с любыми БД. Назвать систему Delphi обычной СУБД мешает, наверное, только то, что, с одной стороны, у нее нет "своего" формата таблиц (языка описания данных), поэтому она использует форматы таблиц других СУБД, например, dBase, Paradox или InterBase (это вряд ли является недостатком, поскольку названные форматы хорошо себя зарекомендовали); с другой стороны, в плане создания приложений различного назначения, в том числе приложений БД, возможности Delphi не уступают возможностям специализированных СУБД, а зачастую и превосходят их.

Модели данных

База данных содержит данные, используемые какой-либо прикладной информационной системой (например, системами "Сирена" или "Экспресс" продажи авиа- и железнодорожных билетов).

В зависимости от вида организации данных различают следующие основные модели представления данных в базе:

- иерархическую;
- сетевую;
- реляционную;
- объектно-ориентированную.

В *иерархической* модели данные представляются в виде древовидной (иерархической) структуры. Подобная организация данных удобна для работы с иерархически упорядоченной информацией, однако при оперировании данными со сложными логическими связями иерархическая модель оказывается слишком громоздкой.

В *сетевой* модели данные организуются в виде произвольного графа. Недостатком сетевой модели является жесткость структуры и высокая сложность ее реализации.

Кроме того, значительным недостатком иерархической и сетевой моделей является то, что структура данных задается на этапе проектирования БД и не может быть изменена при организации доступа к данным.

В *объектно-ориентированной* модели отдельные записи базы данных представляются в виде объектов. Между записями базы данных и функциями их обработки устанавливаются взаимосвязи с помощью механизмов, подобных соответствующим средствам объектно-ориентированных языков программирования. Объектно-ориентированные модели сочетают особенности сетевой и реляционной моделей и используются для создания крупных БД со сложными структурами данных.

Реляционная модель, предложенная в 70-х годах XX века сотрудником фирмы IBM Эдгаром Коддом, получила название от английского термина relation (отношение). Реляционная БД представляет собой совокупность таблиц, *связанных отношениями*. Достоинствами реляционной модели данных являются простота, гибкость структуры, удобство реализации на компьютере, наличие теоретического описания. Большинство современных БД для персональных компьютеров являются реляционными. При последующем изложении материала речь пойдет именно о реляционных БД.

Базы данных и приложения

В зависимости от взаимного расположения приложения и БД можно выделить:

- локальные БД;
- удаленные БД.

Для выполнения операций с локальными БД разрабатываются и используются так называемые *локальные приложения*, а для операций с удаленными БД — *клиент-серверные приложения*.

Расположение БД в значительной степени влияет на разработку приложения, обрабатывающего содержащиеся в этой базе данные.

Так, различают следующие виды приложений:

- приложения, использующие локальные базы данных, называют *одноуровневыми* (однозвенными) приложениями, поскольку приложение и базы данных образуют единую файловую систему;
- приложения, использующие удаленные базы данных, разделяют на двухуровневые (двузвенные) и многоуровневые (многозвенные). *Двухуровневые* приложения содержат клиентскую и серверную части;
- *многоуровневые* (обычно трехуровневые) приложения кроме клиентской и серверной частей имеют дополнительные части. К примеру, в трехуровневых приложениях имеются клиентская часть, сервер приложений и сервер базы данных.

Одно- и двухуровневые приложения Delphi могут осуществлять доступ к локальным и удаленным БД с использованием следующих механизмов:

- BDE (Borland Database Engine — процессор баз данных фирмы Borland), предоставляющий развитый интерфейс API для взаимодействия с базами данных;
- ADO (ActiveX Data Objects — объекты данных ActiveX) осуществляет доступ к информации с помощью OLE DB (Object Linking and Embedding Data Base — связывание и внедрение объектов баз данных);
- dbExpress обеспечивает быстрый доступ к информации в базах данных с помощью набора драйверов;
- InterBase реализует непосредственный доступ к базам данных InterBase.

Выбор варианта технологии доступа к информации в базах данных, кроме прочих соображений, определяется с учетом удобства подготовки разработанного приложения к распространению, а также дополнительного расхода ресурсов памяти. К примеру, инсталляция для BDE требует примерно 15 Мбайт внешней памяти на диске и настройки псевдонимов используемых баз данных. Вариант InterBase вряд ли можно назвать конкурентоспособным, поскольку он ориентирован строго на работу с одноименным сервером баз данных.

Трехуровневые приложения Delphi 7 можно создавать с помощью механизма DataSnap. Используемые при создании трехуровневых (многоуровневых) приложений баз данных компоненты расположены на страницах **DataSnap** и **Data Access** Палитры компонентов.

BDE

BDE представляет собой совокупность динамических библиотек и драйверов, обеспечивающих доступ к данным. Процессор BDE должен устанавливаться на всех компьютерах, на которых выполняются Delphi-приложения, осуществляющие работу с БД. Приложение через BDE передает запрос к базе данных, а обратно получает требуемые данные. Механизм BDE до последней версии системы Delphi получил самое широкое распространение ввиду широкого спектра предоставляемых им возможностей. Идеологи фирмы Borland планируют отказаться от его поддержки, заменив его механизмом dbExpress. Мы приводим множество примеров и описание технологии применения BDE для работы с базами данных в связи с тем, что накоплено большое количество приложений с использованием этого подхода.

ADO

Механизм ADO доступа к информации базы данных является стандартом фирмы Microsoft. Использование этой технологии подразумевает применение настраиваемых провайдеров данных. Технология ADO обеспечивает универсальный механизм доступа из приложений к информации источников данных. Эта технология основана на стандартных интерфейсах COM, являющихся системным механизмом Windows. Это позволяет удобно распространять приложения баз данных без вспомогательных библиотек.

dbExpress

Механизм доступа dbExpress подразумевает использование совокупности драйверов, компонентов, инкапсулирующих соединения, транзакций, запросов, наборов данных и интерфейсов, с помощью которой обеспечивается универсальный доступ к функциям этого механизма. Обеспечение взаимодействия с серверами баз данных по технологии dbExpress основано на использовании специализированных драйверов. Последние для получения данных применяют запросы SQL. На стороне клиента при этом нет кэширования данных, здесь применяются только однонаправленные курсоры и не обеспечивается возможность прямого редактирования наборов данных.

Варианты архитектуры для BDE

Здесь мы рассмотрим различные варианты архитектуры информационной системы на примере технологии BDE. Варианты архитектуры для других технологий доступа к данным рассмотрим позже при непосредственном их описании.

Локальные БД располагаются на том же компьютере, что и работающие с ними приложения. В этом случае говорят, что информационная система имеет локальную архитектуру (рис. 13.1). Работа с БД происходит, как правило, в *однопользовательском* режиме. При необходимости можно запустить на компьютере другое приложение, одновременно осуществляющее доступ к этим же данным. Для управления совместным доступом к БД необходимы специальные средства контроля и защиты. Эти средства могут понадобиться, например, в случае, когда приложение пытается изменить запись, которую редактирует другое приложение. Каждая разновидность БД осуществляет подобный контроль своими способами и обычно имеет встроенные средства разграничения доступа.

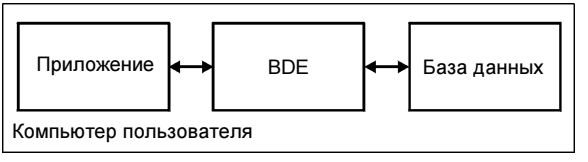


Рис. 13.1. Локальная архитектура

Для доступа к локальной БД процессор баз данных BDE использует стандартные драйверы, которые позволяют работать с форматами БД dBase, Paradox, FoxPro, а также с текстовыми файлами.

При использовании локальной БД в сети можно организовать многопользовательский доступ. В этом случае файлы БД и предназначенное для работы с ней приложение располагаются на сервере сети. Каждый пользователь запускает со своего компьютера это расположенное на сервере приложение, при этом у него запускается копия приложения. Такой сетевой вариант использования локальной БД соответствует архитектуре "файл-сервер". Приложение при использовании архитектуры "файл-сервер" также может быть записано на каждый компьютер сети, в этом случае приложению отдельного компьютера должно быть известно местонахождение общей БД (рис. 13.2).

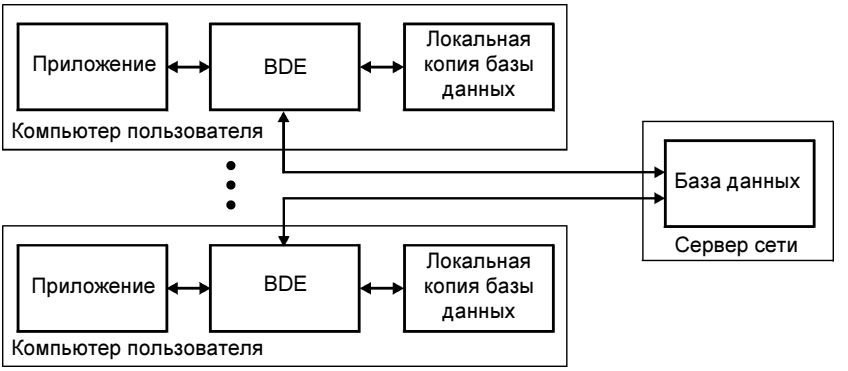


Рис. 13.2. Архитектура "файл-сервер"

При работе с данными на каждом пользовательском компьютере сети используется локальная копия БД. Эта копия периодически обновляется данными, содержащимися в БД на сервере.

Архитектура "файл-сервер" обычно применяется в сетях с небольшим количеством пользователей, для ее реализации подходят персональные СУБД, например, Paradox или dBase. Достоинствами этой архитектуры являются простота реализации, а также то, что приложение фактически разрабатывается в расчете на одного пользователя и не зависит от компьютера сети, на который оно устанавливается.

Однако архитектура "файл-сервер" имеет и существенные недостатки.

- Пользователь работает со своей локальной копией БД, данные в которой обновляются при каждом запросе к какой-либо из таблиц. При этом с сервера пересылается новая копия всей таблицы, данные которой затребованы. Таким образом, если пользователю необходимо несколько записей таблицы, с сервера по сети пересылается вся таблица. В результате циркуляции в сети больших объемов избыточной информации *резко возрастает нагрузка на сеть*, что приводит к соответствующему снижению ее быстродействия и производительности информационной системы в целом.
- В связи с тем, что на каждом компьютере имеется своя копия БД, изменения, сделанные в ней одним пользователем, в течение некоторого времени являются неизвестными другим пользователям. Поэтому требуется постоянное обновление БД. Кроме того, возникает *необходимость синхронизации* работы отдельных пользователей, связанная с блокировкой в таблицах записей, которые в данный момент редактирует другой пользователь.
- Управление БД осуществляется с разных компьютеров, поэтому в значительной степени затруднена *организация управления доступом*, соблюдения *конфиденциальности* и поддержания *целостности* БД.

Удаленная БД размещается на компьютере-сервере сети, а приложение, осуществляющее работу с этой БД, находится на компьютере пользователя. В этом случае мы имеем дело с архитектурой "клиент-сервер" (рис. 13.3), когда информационная система делится на неоднородные части — сервер и клиент БД. В связи с тем, что компьютер-сервер отделен от клиента, его также называют *удаленным сервером*.

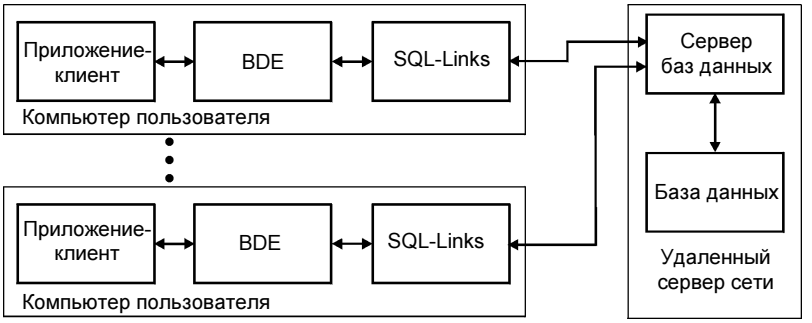


Рис. 13.3. Архитектура "клиент-сервер" ("толстый" клиент)

Клиент — это приложение пользователя. Для получения данных клиент формирует и отсылает запрос удаленному серверу, на котором размещена БД. Запрос формулируется на языке SQL, который является стандартным средством доступа к серверу при использовании реляционных моделей данных. После получения запроса удаленный сервер направляет его программе SQL Server (серверу баз данных) — специальной программе, управляющей удаленной БД и обеспечивающей выполнение запроса и выдачу его результатов клиенту.

Таким образом, в архитектуре "клиент-сервер" клиент посылает запрос на предоставление данных и получает только те данные, которые действительно были затребованы. Вся обработка запроса выполняется на удаленном сервере. Такая архитектура обладает следующими достоинствами.

- Снижение нагрузки на сеть, поскольку теперь в ней циркулирует только нужная информация.
- Повышение безопасности информации, связанное с тем, что обработка запросов всех клиентов выполняется единой программой, расположенной на сервере. Сервер устанавливает общие для всех пользователей правила использования БД, управляет режимами доступа клиентов к данным, запрещая, в частности, одновременное изменение одной записи различными пользователями.
- Уменьшение сложности клиентских приложений за счет отсутствия в них кода, связанного с контролем БД и разграничением доступа к ней.

Для реализации архитектуры "клиент-сервер" обычно используются многопользовательские СУБД, например, Oracle или Microsoft SQL Server. Подобные СУБД также называют *промышленными*, т. к. они позволяют создать информационную систему организации или предприятия с большим числом пользователей. Промышленные СУБД являются сложными системами и требуют мощной вычислительной техники и соответствующего обслуживания. Обслуживание выполняет специалист (или группа специалистов), называемый *системным администратором БД* (администратором). Основные задачи системного администратора:

- защита БД;
- поддержание целостности БД;
- обучение и подготовка пользователей;

- загрузка данных из других БД;
- тестирование данных;
- резервное копирование и восстановление;
- внесение изменений в информационную систему.

Доступ Delphi-приложения к промышленным СУБД осуществляется через драйверы SQL-Links. Отметим, что при работе с "родной" для Delphi СУБД InterBase можно обойтись без драйверов SQL-Links.

Описанная архитектура является двухуровневой (уровень приложения-клиента и уровень сервера БД). Клиентское приложение также называют *сильным*, или "толстым", клиентом. Дальнейшее развитие данной архитектуры привело к появлению трехуровневого варианта архитектуры "клиент-сервер" (приложение-клиент, сервер приложений и сервер БД) (рис. 13.4).

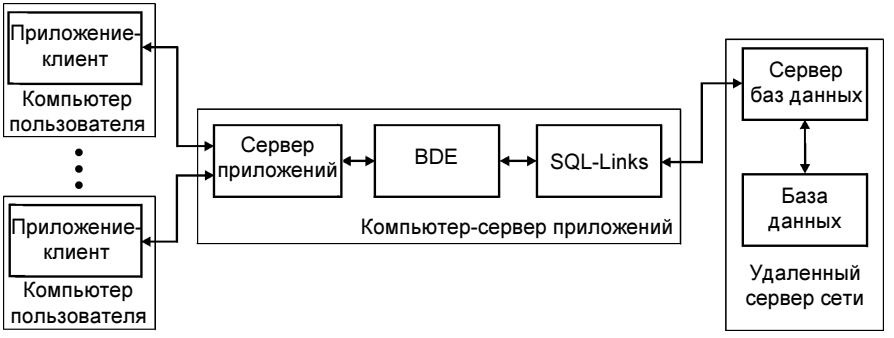


Рис. 13.4. Трехуровневая архитектура "клиент-сервер" ("тонкий" клиент)

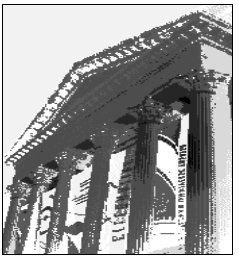
В трехуровневой архитектуре часть средств и кода, предназначенных для организации доступа к данным и их обработке, из приложения-клиента выделяется в сервер приложений. Само клиентское приложение при этом называют *слабым*, или "тонким", клиентом. В сервере приложений удобно располагать средства и код, общие для всех клиентских приложений, например, средства доступа к БД.

Основные достоинства трехуровневой архитектуры "клиент-сервер" состоят в следующем:

- разгрузка сервера от выполнения части операций, перенесенных на сервер приложений;
- уменьшение размера клиентских приложений за счет разгрузки их от лишнего кода;
- единое поведение всех клиентов;
- упрощение настройки клиентов — при изменении общего кода сервера приложений автоматически изменяется поведение приложений-клиентов.

Напомним, что локальные приложения БД называют *одноуровневыми*, а клиент-серверные приложения БД — *многоуровневыми*.

ГЛАВА 14



Реляционные базы данных и средства работы с ними

Большинство современных баз данных для персональных компьютеров являются реляционными. Достоинства реляционной модели организации данных — простота, гибкость структуры, удобство реализации на компьютере, наличие теоретического описания.

Элементы реляционной базы данных

Реляционная база данных (БД) состоит из взаимосвязанных таблиц. Каждая таблица содержит информацию об объектах одного типа, а совокупность всех таблиц образует единую БД.

Таблицы баз данных

Таблицы, образующие БД, находятся в каталоге (папке) на жестком диске. Таблицы хранятся в файлах и похожи на отдельные документы или электронные таблицы (например, табличного процессора Microsoft Excel), их можно перемещать и копировать обычным способом, скажем, с помощью Проводника Windows. Однако в отличие от документов, таблицы БД поддерживают *многопользовательский* режим доступа, это означает, что их могут одновременно использовать несколько приложений.

Для одной таблицы создается несколько файлов, содержащих данные, индексы, ключи и т. п. Главным из них является файл с данными, имя этого файла совпадает с именем таблицы, которое задается при ее создании. В некотором смысле понятия таблицы и ее главного файла являются синонимами, при выборе таблицы выбирается именно ее главный файл: для таблицы dBase это файл с расширением dbf, а для таблицы Paradox — файл с расширением db. Имена остальных файлов таблицы назначаются автоматически — все файлы имеют одинаковые имена, совпадающие с именами таблиц, и разные расширения, указывающие на содержимое соответствующего файла. Расширения файлов приведены в *разд. "Таблицы форматов dBase и Paradox" далее в этой главе*.

Каждая таблица БД состоит из строк и столбцов и предназначена для хранения данных об однотипных объектах информационной системы. Строка таблицы называется

записью, столбец таблицы — *полем* (рис. 14.1). Каждое поле должно иметь уникальное в пределах таблицы имя.

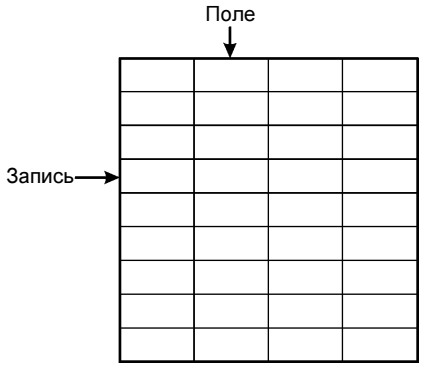


Рис. 14.1. Схема таблицы базы данных

Поле содержит данные одного из допустимых типов, например, строкового, целочисленного или типа "дата". При вводе значения в поле таблицы автоматически производится проверка соответствия типа значения и типа поля. В случае, когда эти типы не совпадают, а преобразование типа значения невозможно, генерируется исключение.

Особенности организации таблиц зависят от конкретной СУБД, используемой для создания и ведения БД. Например, в локальной таблице dBase и в таблице сервера InterBase нет поля автоинкрементного типа (с автоматически наращиваемым значением), а в таблице dBase нельзя определить ключ. Подобные особенности необходимо учитывать при выборе типа (формата) таблицы, т. к. они влияют не только на организацию БД, но и на построение приложения для работы с этой БД. Однако, несмотря на все различия таблиц, существуют общие правила создания и ведения БД, а также разработки приложений, которые и будут далее рассмотрены.

Замечание

В зависимости от типа таблиц и системы разработки приложений может использоваться различная терминология. Например, в InterBase поле таблицы называется столбцом.

Основу таблицы составляет описание ее полей, каждая таблица должна иметь хотя бы одно поле. Понятие структуры таблицы является более широким и включает:

- описание полей;
- ключ;
- индексы;
- ограничения на значения полей;
- ограничения ссылочной целостности между таблицами;
- пароли.

Иногда ограничения на значения полей, ограничения ссылочной целостности между таблицами, а также права доступа называют одним общим термином "ограничения".

Отметим, что отдельные элементы структуры зависят от формата таблиц, например, для таблиц dBase нельзя задать ограничения ссылочной целостности (т. к. у них нет ключей). Все элементы структуры задаются на физическом уровне (уровне таблицы) и действуют для всех программ, выполняющих операции с БД, включая средства разработки и ведения БД (например, программу Database Desktop). Многие из этих элементов (например, ограничения на значения полей или поля просмотра) можно также реализовать в приложении программно, однако в этом случае они действуют только в пределах своего приложения.

С таблицей в целом можно выполнять следующие операции:

- создание (определение структуры);
- изменение структуры (реструктуризация);
- переименование;
- удаление.

При *создании* таблицы задаются структура и имя таблицы. При сохранении на диске создаются все необходимые файлы, относящиеся к таблице. Их имена совпадают с именем таблицы.

При *изменении структуры* таблицы в ней могут измениться имена и характеристики полей, состав и наименования ключа и индексов, ограничения. Однако имена таблицы и ее файлов остаются прежними.

При *переименовании* таблица получает новое имя, в результате чего новое имя также получают все ее файлы. Для этого используются соответствующие программы (утилиты), предназначенные для работы с таблицами БД, например, Database Desktop или Data Pump.

Замечание

Таблицу нельзя переименовать, просто изменив названия всех ее файлов, например, с помощью Проводника Windows.

При *удалении* таблицы с диска удаляются все ее файлы. В отличие от переименования, удаление таблицы можно выполнить посредством любой программы (в том числе и с помощью Проводника Windows).

Ключи и индексы

Ключ представляет собой комбинацию полей, данные в которых однозначно определяют каждую запись в таблице. Простой ключ состоит из одного поля, а составной (сложный) — из нескольких полей. Поля, по которым построен ключ, называют *ключевыми*. В таблице может быть определен только один ключ. Ключ обеспечивает:

- однозначную идентификацию записей таблицы;
- ускорение выполнения запросов к БД;
- установление связи между отдельными таблицами БД;
- использование ограничений ссылочной целостности.

Ключ также называют *первичным ключом* или *первичным (главным) индексом*.

Информация о ключе может храниться в отдельном файле или совместно с данными таблицы. Например, в БД Paradox для этой цели используется отдельный файл (ключевой файл или файл главного индекса) с расширением px. В БД Access вся информация содержится в одном общем файле с расширением mdb. Значения ключа располагаются в определенном порядке. Для каждого значения ключа имеется уникальная ссылка, указывающая на расположение соответствующей записи в таблице (в главном ее файле). Поэтому при поиске записи выполняется не последовательный просмотр всей таблицы, а прямой доступ к записи на основании упорядоченных значений ключа.

Ценой, которую разработчик и пользователь платят за использование такой технологии, является увеличение размера БД вследствие необходимости хранения значений ключа, например, в отдельном файле. Размер этого файла зависит не только от числа записей таблицы (что достаточно очевидно), но и от полей, составляющих ключ. В ключевом файле, кроме ссылок на соответствующие записи таблицы, сохраняются и значения самих ключевых полей. Поэтому при вхождении в состав ключа длинных строковых полей размер ключевого файла может оказаться соизмеримым с размером файла с данными таблицы.

Таблицы различных форматов имеют свои особенности построения ключей. Вместе с тем существуют и общие правила.

- Ключ должен быть уникальным. У составного ключа значения отдельных полей (но не всех одновременно) могут повторяться.
- Ключ должен быть достаточным и неизбыточным, т. е. не содержать поля, которые можно удалить без нарушения уникальности ключа.
- В состав ключа не могут входить поля некоторых типов, например, графическое поле или поле комментария.

Выбор ключевых полей не всегда является простой и очевидной задачей, особенно для таблиц с большим количеством полей. Нежелательно выбирать в качестве ключевых поля, содержащие фамилии людей в таблице сотрудников организации или названия товаров в таблице данных склада. В этом случае высока вероятность существования двух и более однофамильцев, а также товаров с одинаковыми названиями, которые различаются, к примеру, цветом (значение другого поля). Для указанных таблиц можно использовать, например, поле кода сотрудника и поле артикула товара. При этом предполагается, что указанные значения являются уникальными.

Удобным вариантом создания ключа будет использование для него поля соответствующего типа, которое автоматически обеспечивает поддержку уникальности значений. Для таблиц Paradox таким является поле автоинкрементного типа, еще одним достоинством которого является небольшой размер (4 байта). В то же время в таблицах dBase и InterBase поле подобного типа отсутствует, и программист должен обеспечивать уникальность значений ключа самостоятельно, например, используя специальные генераторы.

Отметим, что при создании и ведении БД правильным подходом считается задание в каждой таблице ключа даже в случае, если на первый взгляд он не нужен. В примерах

таблиц, которые приводятся при изложении материала, как правило, ключ создается, и для него вводится специальное автоинкрементное поле с именем `Code` или `Number`.

Индекс, как и ключ, строится по полям таблицы, однако он может допускать повторение значений составляющих его полей — в этом и состоит его основное отличие от ключа. Поля, по которым построен индекс, называют *индексными*. Простой индекс состоит из одного поля, а составной (сложный) — из нескольких полей.

Индексы при их создании именуются. Как и в случае с ключом, в зависимости от СУБД индексы могут храниться в отдельных файлах или совместно с данными. Создание индекса называют *индексированием таблицы*.

Использование индекса обеспечивает:

- увеличение скорости доступа к данным (поиска);
- сортировку записей;
- установление связи между отдельными таблицами БД;
- использование ограничений ссылочной целостности.

В двух последних случаях индекс применяется совместно с ключом второй таблицы.

Как и ключ, индекс представляет собой своеобразное оглавление таблицы, просмотр которого выполняется перед обращением к ее записям. Таким образом, использование индекса повышает *скорость доступа* к данным в таблице за счет того, что доступ выполняется не последовательным, а индексно-последовательным методом.

Сортировка представляет собой упорядочивание записей по полю или группе полей в порядке возрастания или убывания их значений. Можно сказать, что индекс служит для сортировки таблиц по индексным полям. В частности, в Delphi записи набора `Table` можно сортировать только по индексным полям. Набор данных `Query` позволяет выполнить средствами SQL сортировку по любым полям, однако и в этом случае для индексированных полей упорядочивание записей выполняется быстрее.

Для одной таблицы можно создать несколько индексов. В каждый момент времени один из них можно сделать текущим, т. е. активным. Даже при существовании нескольких индексов таблица может не иметь текущего индекса (текущий индекс важен, например, при выполнении поиска и сортировки записей набора данных `Table`).

Ключевые поля обычно автоматически индексируются. В таблицах Paradox ключ также является главным (первичным) индексом, который не именуется. Для таблиц dBase ключ не создается, и его роль выполняет один из индексов.

Замечание

Создание ключа может привести к побочным эффектам. Так, если в таблице Paradox определить ключ, то записи автоматически упорядочиваются по его значениям, что в ряде случаев является нежелательным.

Таким образом, использование ключей и индексов позволяет:

- однозначно идентифицировать записи;
- избегать дублирования значений в ключевых полях;
- выполнять сортировку таблиц;

- ускорять операции поиска в таблицах;
- устанавливать связи между отдельными таблицами БД;
- использовать ограничения ссылочной целостности.

Одной из основных задач БД является обеспечение *быстрого доступа к данным* (поиска данных). Время доступа к данным в значительной степени зависит от используемых для поиска данных методов и способов.

Методы и способы доступа к данным

Выделяют следующие методы доступа к данным таблиц:

- последовательный;
- прямой;
- индексно-последовательный.

При *последовательном* методе выполняется последовательный просмотр всех записей таблицы и поиск нужных из них. Этот метод доступа является крайне неэффективным и приводит к значительным временным затратам на поиск, прямо пропорциональным размеру таблицы (числу ее записей). Поэтому его рекомендуется использовать только для относительно небольших таблиц.

При *прямом* доступе нужная запись выбирается в таблице на основании ключа или индекса. При этом просмотр других записей не выполняется. Напомним, что значения ключей и индексов располагаются в упорядоченном виде и содержат ссылку, указывающую на расположение соответствующей записи в таблице. При поиске записи выполняется не последовательный просмотр всей таблицы, а непосредственный доступ к записи на основании ссылки.

Индексно-последовательный метод доступа включает в себя элементы последовательного и прямого методов доступа и используется при поиске группы записей. Этот метод реализуется только при наличии индекса, построенного по полям, значения которых должны быть найдены. Суть его заключается в том, что находится индекс первой записи, удовлетворяющей заданным условиям, и соответствующая запись выбирается из таблицы на основании ссылки. Это является прямым доступом к данным. После обработки первой найденной записи осуществляется переход к следующему значению индекса, и в таблице выбирается запись, соответствующая значению этого индекса. Так последовательно перебираются индексы всех записей, удовлетворяющих заданным условиям, что является последовательным доступом.

Достоинством прямого и индексно-последовательного методов является максимальная возможная скорость доступа к данным, плата за которую — расход памяти на хранение информации о ключах и индексах.

Указанные методы доступа реализуются СУБД и не требуют специального программирования. Задачей разработчика является определение соответствующей структуры БД, в данном случае — определение ключей и индексов. Так, если для поля создан индекс, то при поиске записей по этому полю автоматически используется индексно-последовательный метод доступа, в противном случае — последовательный метод.

Замечание

При создании составного индекса важен порядок составляющих его полей. Например, если индекс создан по полям "фамилия", "номер отдела" и "дата рождения", а поиск ведется одновременно по полям "фамилия", "дата рождения" и "номер отдела", то такой индекс использован не будет. В результате доступ к таблице осуществляется последовательным методом. В подобной ситуации (для таблицы с большим числом записей) разработчик должен создать также индекс, построенный по полям "фамилия", "дата рождения" и "номер отдела".

При выполнении операций с таблицами используется один из следующих *способов доступа к данным*:

- навигационный;
- реляционный.

Навигационный способ доступа заключается в обработке каждой отдельной записи таблицы. Этот способ обычно используется в локальных БД или в удаленных БД небольшого размера. Если необходимо обработать несколько записей, то все они обрабатываются поочередно.

Реляционный способ доступа основан на обработке сразу *группы записей*, при этом если необходимо обработать одну запись, то обрабатывается группа, состоящая из одной записи. Так как реляционный способ доступа основывается на SQL-запросах, его также называют *SQL-ориентированным*. Этот способ доступа ориентирован на выполнение операций с удаленными БД и является предпочтительным при работе с ними, хотя его можно использовать и для локальных БД.

Способ доступа к данным выбирается программистом и зависит от средств доступа к БД, используемых при разработке приложения. Например, в приложениях, создаваемых в Delphi, реализацию навигационного способа доступа можно осуществить посредством компонентов `Table` или `Query`, а реляционного — с помощью компонента `Query`.

Таким образом, методы доступа к данным определяются структурой БД, а способы доступа — приложением.

Связь между таблицами

В частном случае БД может состоять из одной таблицы, содержащей, например, дни рождения сотрудников организации. Однако обычно реляционная БД состоит из набора взаимосвязанных таблиц. Организация связи (отношений) между таблицами называется *связыванием* или *соединением таблиц*.

Связи между таблицами можно устанавливать как при создании БД, так и при выполнении приложения, используя средства, предоставляемые СУБД. Связывать можно две или несколько таблиц. В реляционной БД, помимо связанных, могут быть и отдельные таблицы, не соединенные ни с одной другой таблицей. Это не меняет сути реляционной БД, которая содержит единую информацию об информационной системе, связанную не в буквальном смысле (связь между таблицами), а в функциональном смысле — вся информация относится к одной системе.

Для связывания таблиц используются *поля связи* (иногда применяется термин "совпадающие поля"). Поля связи обязательно должны быть индексированными. В подчи-

ненной таблице для связи с главной таблицей задается индекс, который также называется *внешним ключом*. Состав полей этого индекса должен полностью или частично совпадать с составом полей индекса главной таблицы.

Особенности использования индексов зависят от формата связываемых таблиц. Так, для таблиц dBase индексы строятся по одному полю и нет деления на ключ (главный или первичный индекс) и индексы. Для организации связи в главной и подчиненной таблицах выбираются индексы, составленные по полям совпадающего типа, например, целочисленного.

Для таблиц Paradox в качестве полей связи главной таблицы должны использоваться поля ключа, а для подчиненной таблицы — поля индекса. Кроме того, в подчиненной таблице обязательно должен быть определен ключ. На рис. 14.2 показана схема связи между таблицами БД Paradox.

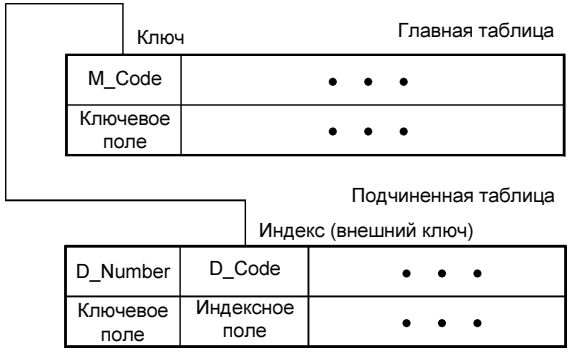


Рис. 14.2. Схема связи между таблицами базы данных Paradox

В главной таблице определен ключ, построенный по полю `M_Code` автоинкрементного типа. В подчиненной таблице определен ключ по полю `D_Number` также автоинкрементного типа и индекс, построенный по полю `D_Code` целочисленного типа. Связь между таблицами устанавливается по полям `D_Code` и `M_Code`. Индекс по полю `D_Code` является внешним ключом. В названия полей включены префиксы, указывающие на принадлежность поля соответствующей таблице. Так, названия полей главной таблицы начинаются с буквы `M` (Master), а названия полей подчиненной таблицы начинаются с буквы `D` (Detail). Подобное именование полей облегчает ориентацию в их названиях, особенно при большом количестве таблиц.

Замечание

Как отмечалось, поля связи должны быть индексированными, хотя, строго говоря, это требование не всегда является обязательным. При доступе к данным средствами языка SQL можно связать (соединить) между собой таблицы и по неиндексированным полям. Однако в этом случае скорость выполнения операций будет низкой.

Связь между таблицами определяет отношение подчиненности, при котором одна таблица является *главной* (родительской, или мастером — Master), а вторая — *подчиненной* (дочерней, или детальной — Detail). Саму связь (отношение) называют связь "главный — подчиненный", "родительский — дочерний" или "мастер — детальный".

Существуют следующие виды связи:

- отношение "один-к-одному";
- отношение "один-ко-многим";
- отношение "много-к-одному";
- отношение "много-ко-многим".

Отношение *"один-к-одному"* означает, что одной записи в главной таблице соответствует одна запись в подчиненной таблице. При этом возможны два варианта:

- для каждой записи главной таблицы есть запись в подчиненной таблице;
- в подчиненной таблице может не быть записей, соответствующих записям главной таблицы.

Отношение "один-к-одному" обычно используется при разбиении таблицы с большим числом полей на несколько таблиц. В этом случае в первой таблице остаются поля с наиболее важной и часто используемой информацией, а остальные поля переносятся в другую (другие) таблицу.

Отношение *"один-ко-многим"* означает, что одной записи главной таблицы в подчиненной таблице может соответствовать несколько записей, в том числе ни одной. Этот вид отношения встречается наиболее часто. После установления связи между таблицами при перемещении на какую-либо запись в главной таблице в подчиненной таблице автоматически становятся доступными записи, у которых значение поля связи равно значению поля связи текущей записи главной таблицы. Такой отбор записей подчиненной таблицы является своего рода фильтрацией.

Типичным примером является, например, организация учета выдачи книг в библиотеке, для которой удобно создать следующие две таблицы:

- таблицу карточек читателей, содержащую такую информацию о читателе, как фамилия, имя, отчество, дата рождения и домашний адрес;
- таблицу выдачи книг, в которую заносится информация о выдаче книги читателю и о возврате книги.

В этой ситуации главной является таблица карточек читателей, а подчиненной — таблица выдачи книг. Один читатель может иметь на руках несколько книг, поэтому одной записи в главной таблице может соответствовать несколько записей в подчиненной таблице. Если читатель сдал все книги или еще не брал ни одной книги, то для него в подчиненной таблице записей нет. После связывания обеих таблиц при выборе записи с данными читателя в таблице выдачи книг будут доступны только записи с данными о книгах, находящихся на руках этого читателя.

В приведенном примере предполагается, что после возврата книги соответствующая ей запись удаляется из таблицы выдачи книг. Вместо удаления записи можно заносить в соответствующее поле дату возврата книги.

Отношение *"много-к-одному"* отличается от отношения "один-ко-многим" только направлением. Если на отношение "один-ко-многим" посмотреть со стороны подчиненной таблицы, а не главной, то оно превращается в отношение "много-к-одному".

Отношение *"много-ко-многим"* имеет место, когда одной записи главной таблицы может соответствовать несколько записей подчиненной таблицы и одновременно одной

записи подчиненной таблицы — несколько записей главной таблицы. Подобное отношение реализуется, например, при планировании занятий в институте и устанавливается между таблицами, в которых хранится информация о номерах аудиторий и номерах учебных групп. Так как учебная группа может заниматься в разных аудиториях, то одной записи об учебной группе (первая таблица) может соответствовать несколько записей о занимаемых этой группой аудиториях. В то же время в одной аудитории могут заниматься разные учебные группы (даже одновременно), поэтому одной записи об аудитории может соответствовать несколько записей об учебных группах (вторая таблица).

На практике отношение "много-ко-многим" используется достаточно редко. Причинами являются сложность организации связи между таблицами и взаимодействия между их записями. Кроме того, многие СУБД, в том числе Paradox, не поддерживают организацию ссылочной целостности для подобного отношения. Отметим также, что для отношения "много-ко-многим" понятия главной и подчиненной таблицы не имеют смысла.

Среди рассмотренных отношений наиболее общим является отношение "один-ко-многим". Другие виды отношений можно рассматривать как его варианты — отношение "один-к-одному" представляет собой частный случай этого отношения, а отношение "много-к-одному" является его "переворотом". Отношение "много-ко-многим" можно свести к отношению "один-ко-многим", соответствующим образом преобразовав и разделив таблицы. В дальнейшем предполагается, что таблицы связаны именно отношением "один-ко-многим".

Работа со связанными таблицами имеет следующие особенности.

- При изменении (редактировании) поля связи может нарушиться связь между записями двух таблиц. Поэтому при редактировании поля связи записи главной таблицы нужно соответственно изменять и значения поля связи всех подчиненных таблиц.
- При удалении записи главной таблицы нужно удалять и соответствующие ей записи в подчиненной таблице (каскадное удаление).
- При добавлении записи в подчиненную таблицу значение ее поля связи должно быть установлено равным значению поля связи главной таблицы.

Ограничения по установке, изменению полей связи и каскадному удалению записей могут быть наложены на таблицы при их создании. Эти ограничения, наряду с другими элементами, например описаниями полей и индексов, входят в структуру таблицы и действуют для всех приложений, которые выполняют операции с БД. Указанные ограничения можно задать при создании или реструктуризации таблицы, например, в среде программы Database Desktop, которая позволяет устанавливать связи между таблицами при их создании.

Ограничения, связанные с установкой, изменением значений полей связи и каскадным удалением записей, могут и не входить в структуру таблицы (таблиц), а реализовываться программным способом. В этом случае программист должен обеспечить:

- организацию связи между таблицами;
- установку значения поля связи подчиненной таблицы (это может также выполняться автоматически);

- контроль (запрет) редактирования полей связи;
- организацию (запрет) каскадного удаления записей.

Например, в случае удаления записи из главной таблицы программист должен проверить наличие соответствующих записей в подчиненной таблице. Если такие записи есть, то необходимо удалить и их или, наоборот, запретить удаление записей из обеих таблиц. И в том, и в другом случае пользователю должно быть выдано предупреждение.

Механизм транзакций

Информация БД в любой момент времени должна быть целостной и непротиворечивой. Одним из путей обеспечения этого является использование механизма транзакций.

Транзакция представляет собой выполнение последовательности операций. При этом возможны две ситуации.

- Успешно завершены все операции. В этом случае транзакция считается успешной, и все изменения в БД, которые были произведены в рамках транзакции отдельными операциями, подтверждаются. В результате БД переходит из одного целостного состояния в другое.
- Неудачно завершена хотя бы одна операция. При этом вся транзакция считается неуспешной, и результаты выполнения всех операций (даже успешно выполненных) отменяются. В результате происходит возврат БД в состояние, в котором она находилась до начала транзакции.

Таким образом, успешная транзакция переводит БД из одного целостного состояния в другое. Использование механизма транзакций необходимо:

- при выполнении последовательности взаимосвязанных операций с БД;
- при многопользовательском доступе к БД.

Транзакция может быть неявной или явной. *Неявная* транзакция стартует автоматически, а по завершении также автоматически подтверждается или отменяется. *Явной* транзакцией управляет программист с использованием компонента `Database` и/или средств SQL.

Часто в транзакцию объединяются операции над несколькими таблицами, когда производятся действия по внесению в разные таблицы взаимосвязанных изменений. Пусть осуществляется перенос записей из одной таблицы в другую. Если запись сначала удаляется из первой таблицы, а затем заносится во вторую таблицу, то при сбое, например из-за перерыва в энергопитании компьютера, возможна ситуация, когда запись уже удалена, но во вторую таблицу не попала. Если запись сначала заносится во вторую таблицу, а потом удаляется из первой таблицы, то при сбое возможна ситуация, когда запись будет находиться в двух таблицах. В обоих случаях имеет место нарушение целостности и непротиворечивости БД.

Для предотвращения подобной ситуации операции удаления записи из одной таблицы и занесения ее в другую таблицу объединяются в одну транзакцию. Выполнение этой транзакции гарантирует, что при любом ее результате целостность БД нарушена не будет.

Еще одним примером, демонстрирующим необходимость применения механизма транзакций, является складской учет товара. При поступлении товара на склад в таблицу движения товара заносится запись с данными о названии, количестве товара и дате его поступления. Затем в таблице склада соответственно количеству поступившего товара увеличивается наличное количество этого товара. При возникновении какой-либо ошибки, связанной с записью наличного количества товара, новое значение может быть не занесено в соответствующую запись, в результате чего будет нарушена целостность БД, и она будет содержать некорректные значения. Такая ситуация возможна, например, в случае многопользовательского доступа к БД при редактировании этой записи другим приложением. Поэтому в случае невозможности внести изменения в наличное количество товара должно блокироваться и добавление новой записи в таблицу движения товара.

Для реализации механизма транзакций СУБД предоставляют соответствующие средства.

Бизнес-правила

Бизнес-правила представляют собой механизмы управления БД и предназначены для поддержания БД в целостном состоянии, а также для выполнения ряда других действий, например, накапливания статистики работы с БД.

Замечание

В данном контексте бизнес-правила являются просто правилами управления БД и не имеют отношения к бизнесу как предпринимательству.

В первую очередь бизнес-правила реализуют следующие ограничения БД:

- задание допустимого диапазона значений;
- задание значения по умолчанию;
- требование уникальности значения;
- запрет пустого значения;
- ограничения ссылочной целостности.

Бизнес-правила можно реализовывать как на физическом, так и на программном уровнях. В первом случае эти правила (например, ограничения ссылочной целостности для связанных таблиц) задаются при создании таблиц и входят в структуру БД. В дальнейшей работе нельзя нарушить или обойти ограничение, заданное на физическом уровне.

Вместо заданных на физическом уровне бизнес-правил или в дополнение к ним можно определить бизнес-правила на программном уровне. Действие этих правил распространяется только на приложение, в котором они реализованы. Для программирования в приложении бизнес-правил используются компоненты и предоставляемые ими средства. Достоинство такого подхода заключается в легкости изменения бизнес-правил и определении правил "своего" приложения. Недостатком является снижение безопасности БД, связанное с тем, что каждое приложение может устанавливать свои правила управления БД. В *главе 19*, посвященной навигационному способу доступа, приводится

пример программирования бизнес-правил в приложении, связанный с каскадным удалением записей связанных таблиц.

При работе с удаленными БД в архитектуре "клиент-сервер" бизнес-правила можно реализовывать также на сервере.

Словарь данных

Словарь данных представляет собой совокупность определений БД и атрибутов. Словарь данных позволяет сформировать и сохранить характеристики, которые в дальнейшем можно использовать для описания БД.

Использование словаря данных позволяет:

- ускорить процесс создания БД;
- облегчить изменение характеристик БД;
- придать единообразный вид визуальным компонентам приложения.

Определение БД является специализированной БД, которая описывает структуру базы, но не содержит данных. Это описание структуры можно использовать для создания других БД.

Атрибуты представляют собой совокупности характеристик отдельных полей. Заданные через атрибуты характеристики поля при выполнении приложения устанавливаются в качестве значений соответствующих свойств визуальных компонентов, которые отображают значения поля, например, `DBGrid` или `DBText`. Приведем некоторые наиболее распространенные *характеристики полей* (они же *свойства* визуальных компонентов):

- `Alignment` — выравнивание;
- `DisplayLabel` — заголовок столбца (сетки `DBGrid`);
- `ReadOnly` — недоступность поля для редактирования (поле предназначено только для чтения);
- `Required` — требование обязательного ввода значения;
- `Visible` — видимость;
- `DisplayFormat` — формат отображаемого значения;
- `MinValue` — минимальное значение;
- `MaxValue` — максимальное значение.

Для работы со словарем данных удобно использовать программу SQL Explorer.

Таблицы форматов dBase и Paradox

Delphi не имеет своего формата таблиц, но поддерживает как собственные два типа локальных таблиц — dBase и Paradox. Каждая из этих таблиц имеет свои особенности.

Таблицы dBase являются одним из первых появившихся форматов таблиц для персональных компьютеров и поддерживаются многими системами, которые связаны с разработкой и обслуживанием приложений, работающих с БД.

Основные достоинства таблиц dBase:

- простота использования;
- совместимость с большим числом приложений.

В табл. 14.1 содержится список типов полей таблиц dBase IV. Для каждого типа приводится символ, используемый для его обозначения в программе Database Desktop (это программа создания и редактирования таблиц, SQL-запросов и запросов QBE), а также описание значений, которые может содержать поле рассматриваемого типа.

Таблица 14.1. *Типы полей таблиц dBase IV*

Тип	Обозначение	Описание значения
`Character`	`C`	Строка символов. Длина не более 255 символов
`Float`	`F`	Число с плавающей точкой. Диапазон -10^{308} ... 10^{308}. Точность 15 цифр мантиссы
`Number`	`N`	Число в двоично-десятичном формате BCD (Binary Coded Decimal)
`Date`	`D`	Дата
`Logical`	`L`	Логическое значение. Допустимы значения `True` (истина) и `False` (ложь). Разрешается использование прописных букв
`Memo`	`M`	Строка символов. Длина не ограничена. Символы хранятся в файле с расширением dbt
`OLE`	`O`	Данные в формате, который поддерживается технологией OLE. Данные хранятся в файле с расширением dbt
`Binary`	`B`	Последовательность байтов. Длина не ограничена. Байты содержат произвольное двоичное значение, хранятся в файле с расширением dbt

Замечание

При работе с таблицей в среде программы Database Desktop значения полей типа `Binary`, `Memo` и `OLE` не отображаются.

Таблицы dBase являются достаточно простыми и используют для своего хранения на дисках относительно мало физических файлов. По расширению файлов можно определить, какие данные они содержат.

- dbf — таблица с данными.
- dbt — данные больших двоичных объектов, или BLOB-данные (Binary Large OBject). К ним относятся двоичные, Memo- и OLE-поля. Memo-поле также называют полем комментариев.
- mdx — поддерживаемые индексы.
- ndx — индексы, непосредственно не поддерживаемые форматом dBase. При использовании таких индексов программист должен обрабатывать их самостоятельно.

Имя поля в таблице dBase должно состоять из букв и цифр и начинаться с буквы. Максимальная длина имени составляет 10 символов. В имена нельзя включать специальные символы и пробел.

К недостаткам таблиц dBase относится то, что они не поддерживают автоматическое использование парольной защиты и контроль целостности связей, поэтому программист должен кодировать эти действия самостоятельно.

Таблицы Paradox являются достаточно развитыми и удобными для создания БД. Можно отметить следующие их достоинства:

- большое количество типов полей для представления данных различных типов;
- поддержка целостности данных;
- организация проверки вводимых данных;
- поддержка парольной защиты таблиц.

Большой набор типов полей позволяет гибко выбирать тип для точного представления данных, хранимых в базе. Например, для представления числовой информации можно использовать один из пяти числовых типов. В табл. 14.2 содержится список типов полей для таблиц Paradox 7. Для каждого типа приводятся символ, используемый для обозначения этого типа в программе Database Desktop, и описание значений, которые может содержать поле рассматриваемого типа.

Таблица 14.2. *Типы полей таблиц в Paradox 7*

Тип	Обозначение	Описание значения
`Alpha`	`A`	Строка символов. Длина не более 255 символов
`Number`	`N`	Число с плавающей точкой. Диапазон -10^{307} ... 10^{308}. Точность 15 цифр мантиссы
`Money`	`$`	Денежная сумма. Отличается от типа `Number` тем, что в значении отображается денежный знак. Обозначение денежного знака зависит от установок Windows
`Short`	`S`	Целое число. Диапазон –32 768 ... 32 767
`LongInteger`	`I`	Целое число. Диапазон –2 147 483 648 ... 2 147 483 647
`BCD`	`#`	Число в двоично-десятичном формате
`Date`	`D`	Дата. Диапазон 01.01.9999 до н. э. ... 31.12.9999
`Time`	`T`	Время
`Timestamp`	`@`	Дата и время
`Memo`	`M`	Строка символов. Длина не ограничена. Первые 240 символов хранятся в файле таблицы, остальные в файле с расширением mb
`Formatted Memo`	`F`	Строка символов. Отличается от типа `Memo` тем, что строка может содержать форматированный текст
`Graphic`	`G`	Графическое изображение. Форматы BMP, PCX, TIFF, GIF и EPS. При загрузке в поле изображение преобразуется к формату BMP. Для хранения изображения используется файл с расширением mb

Таблица 14.2 *(окончание)*

Тип	Обозначение	Описание значения
OLE	O	Данные в формате, который поддерживается технологией OLE. Данные хранятся в файле с расширением mb
Logical	L	Логическое значение. Допустимы значения True (истина) и False (ложь). Разрешается использование прописных букв
Autoincrement	+	Автоинкрементное поле. При добавлении к таблице новой записи в поле автоматически заносится значение, на единицу большее, чем в последней добавленной записи. При удалении записи значение ее автоинкрементного поля больше не будет использовано. Значение автоинкрементного поля доступно для чтения и обычно используется в качестве ключевого поля
Binary	B	Последовательность байтов. Длина не ограничена. Байты содержат произвольное двоичное значение. Первые 240 байтов хранятся в файле таблицы, остальные в файле с расширением mb
Bytes	Y	Последовательность байтов. Длина не более 255 байтов

Замечание

При работе с таблицей в среде программы Database Desktop значения полей типа Graphic, Binary, Memo и OLE не отображаются.

Имя поля в таблице Paradox должно состоять из букв (допускается кириллица) и цифр и начинаться с буквы. Максимальная длина имени составляет 25 символов. В имени можно использовать такие символы, как пробел, #, $ и некоторые другие. Не рекомендуется использовать символы ., ! и |, т. к. они зарезервированы в Delphi для других целей.

При задании ключевых полей они должны быть первыми в структуре таблицы.

Замечание

Если требуется обеспечить перенос или совместимость данных из таблиц Paradox с таблицами других форматов, желательно выбирать имя поля длиной не более 10 символов и составлять его из латинских букв и цифр.

Поддержка концепции целостности данных обеспечивает правильность ссылок между таблицами. Например, если в БД имеются таблицы клиентов и заказов, то эти таблицы могут быть связаны следующим образом: каждая запись таблицы заказов ссылается через индексное поле на запись в таблице клиентов, соответствующую сделавшему заказ клиенту. Если в таблице клиентов любым способом удалить запись с информацией о каком-либо клиенте, то BDE автоматически удалит все записи, соответствующие этому клиенту, и из таблицы заказов. Подобное удаление взаимосвязанных записей называют *каскадным*.

Для полей можно определить специальный диапазон, в котором должны находиться вводимые в них значения. Кроме того, для каждого поля можно определить минималь-

ное и максимальное допустимые значения. При попытке ввода в поле значения, выходящего за допустимые границы, возникает исключение, значение не вводится и содержимое поля не изменяется. Например, для поля `Salary` (оклад) в качестве минимального значения логично указать ноль, тем самым в это поле запрещается вводить отрицательные значения. Максимальное значение поля `Salary` зависит от организации, страны и от других факторов.

Наряду с диапазоном допустимых значений для каждого поля можно задать значение по умолчанию, которое автоматически заносится в поле при добавлении к таблице новой записи.

При работе с конфиденциальной информацией может потребоваться защита таблиц и их полей. Для каждой таблицы Paradox следует указать основной пароль, который используется при изменениях во всей таблице, связанных со сменой структуры или с редактированием данных в любом поле. Возможно также задание дополнительного пароля, позволяющего ограничить доступ к конкретному полю или группе полей таблицы, а также набора операций, применимых к таблице (например, разрешения только на чтение записей таблицы без возможности их модификации). После установки паролей они будут автоматически запрашиваться и контролироваться при любой попытке доступа к таблице.

Определенным недостатком таблиц Paradox является наличие относительно большого количества типов файлов, требуемых для хранения содержащихся в таблице данных. При копировании или перемещении какой-либо таблицы из одного каталога в другой необходимо обеспечить копирование или перемещение всех файлов, относящихся к этой таблице. Файлы таблиц Paradox имеют следующие расширения:

- db — таблица с данными;
- mb — BLOB-данные;
- px — главный индекс (ключ);
- xg* и yg* — вторичные индексы;
- val — параметры для проверки данных и целостности ссылок;
- tv и fam — форматы вывода таблицы в программе Database Desktop.

Замечание

Указанные файлы создаются, только если в них есть необходимость; конкретная таблица может не иметь всех приведенных файлов.

Кроме названных файлов, при работе в сети для управления доступом к таблицам Paradox используются файлы с расширением net.

Средства для работы с реляционными базами данных

Хотя система Delphi не имеет своего формата таблиц БД, она тем не менее обеспечивает мощную поддержку большого количества различных СУБД — как локальных (например, dBase или Paradox), так и промышленных (например, Sybase или InterBase).

Средства Delphi, предназначенные для работы с БД, можно разделить на два вида:

- инструменты;
- компоненты.

К *инструментам* относятся специальные программы и пакеты, обеспечивающие обслуживание БД вне разрабатываемых приложений.

Компоненты предназначены для создания приложений, осуществляющих операции с БД.

Напомним, что в Delphi имеется окно Обозревателя дерева объектов, которое отображает иерархическую структуру объектов текущей формы. При разработке приложений баз данных это окно удобно использовать для просмотра структуры базы данных и изменения связей между компонентами. Кроме того, в окне Редактора кода имеется вкладка **Diagram**, служащая для отображения и настройки взаимосвязей между элементами баз данных.

Инструменты

Для операций с БД система Delphi предлагает следующий набор инструментов.

- Borland Database Engine (BDE) — процессор баз данных, который представляет собой набор динамических библиотек и драйверов, предназначенных для организации доступа к БД из Delphi-приложений. BDE является центральным звеном при организации доступа к данным.
- BDE Administrator — утилита для настройки различных параметров BDE, настройки драйверов баз данных, создания и удаления драйверов ODBC, создания и обслуживания псевдонимов.
- Database Desktop — программа создания и редактирования таблиц, SQL-запросов и запросов QBE.
- SQL Explorer — Проводник БД, позволяющий просматривать и редактировать БД и словари данных.
- SQL Builder — программа визуального конструирования SQL-запросов.
- SQL Monitor — программа отслеживания порядка выполнения SQL-запросов к удаленным БД.
- Data Pump — программа для переноса данных (схемы базы данных и содержимого) между БД.
- IBConsole — программа для управления удаленными БД.
- InterBase Server Manager — программа для запуска сервера InterBase.
- SQL Links — драйверы для доступа к удаленным промышленным СУБД, таким как Microsoft SQL Server или Oracle. К промышленному серверу InterBase, поставляемому с Delphi, доступ также можно организовать напрямую через BDE, не используя драйвер SQL Links. В Delphi 7 вместо SQL Links для доступа к базам данных SQL Server рекомендуется использовать dbExpress.

- dbExpress — набор драйверов для доступа к базам данных SQL с помощью таких компонентов, как `SQLConnection`, `SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`. dbExpress включает в свой состав следующие драйверы:
  - InterBase (файл dbexpint.dll);
  - DB2 (файл dbexpdb2.dll);
  - Oracle (файл dbexpora.dll);
  - MSSQL (файл dbexpmss.dll);
  - MySQL (файл dbexpmys.dll).
- InterBase Server — клиентская и серверная части сервера InterBase.

Одни инструменты, например, BDE Administrator и SQL Explorer, можно использовать для работы с локальными и удаленными БД, другие, например, IBConsole, — для работы с удаленными БД.

Компоненты

Рассмотрим компоненты, используемые для создания приложений БД. Кроме компонентов, Delphi предоставляет разработчику специальные объекты, например, объекты типа `Field`. Как и другие элементы управления Delphi, связанные с БД компоненты делятся на визуальные и невизуальные.

Невизуальные компоненты предназначены для организации доступа к данным, содержащимся в таблицах. Они представляют собой промежуточное звено между данными таблиц БД и визуальными компонентами.

Визуальные компоненты используются для создания интерфейсной части приложения. С их помощью пользователь может выполнять такие операции с таблицами БД, как просмотр или редактирование данных. Визуальные компоненты также называют *элементами, чувствительными к данным*.

Компоненты, используемые для работы с БД, находятся на страницах **Data Access**, **Data Controls**, **dbExpress**, **DataSnap**, **BDE**, **ADO**, **InterBase**, **Decision Cube**, **Rave** и **InterBase Admin** Палитры компонентов. Некоторые компоненты предназначены специально для работы с удаленными БД в архитектуре "клиент-сервер". Отметим, что в Палитре компонентов Delphi 7 появились новые страницы и по сравнению с предыдущими версиями изменилось распределение компонентов, используемых для работы с БД, по страницам. В частности, вместо страницы **QReport** появилась страница **Rave**.

Замечание

Состав компонентов можно настроить в диалоговом окне **Palette Properties** (Свойства палитры), вызываемом командой **Properties** (Свойства) контекстного меню Палитры компонентов. Здесь приведен состав Палитры компонентов, который получается после установки Delphi 7.

На странице **Data Access** (рис. 14.3) находятся невизуальные компоненты, предназначенные для организации доступа к данным:

- `DataSource` (источник данных);
- `ClientDataSet` (клиентский набор данных);

- DataSetProvider (провайдер набора данных);
- XML Transform (преобразователь документа XML в пакет данных и обратно);
- XML TransformProvider (провайдер данных для преобразования документа XML);
- XML TransformClient (адаптер между документом XML и провайдером).

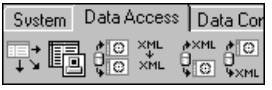


Рис. 14.3. Страница **Data Access**

Рис. 14.4. Страница **Data Controls**

На странице **Data Controls** (рис. 14.4) расположены визуальные компоненты, предназначенные для управления данными:

- DBGrid (сетка, или таблица);
- DBNavigator (навигационный интерфейс);
- DBText (надпись);
- DBEdit (однострочный редактор, или поле редактирования);
- DBMemo (многострочный редактор, или панель редактирования);
- DBImage (графическое изображение);
- DBListBox (список);
- DBComboBox (комбинированный список);
- DBCheckBox (флажок);
- DBRadioGroup (группа переключателей);
- DBLookupListBox (список, формируемый по полю другого набора данных);
- DBLookupComboBox (комбинированный список, формируемый по полю другого набора данных);
- DBRichEdit (полнофункциональный тестовый редактор, или поле редактирования);
- DBCtrlGrid (модифицированная сетка);
- DBChart (диаграмма).

На странице **dbExpress** (рис. 14.5) размещены компоненты, предназначенные для соединения приложений с базами данных с помощью bdExpress:

- SQLConnection (инкапсуляция bdExpress-соединения с сервером БД);
- SQLDataSet (однонаправленный набор данных);
- SQLQuery (однонаправленный набор данных Query);
- SQLStoredProc (вызов хранимой процедуры сервера);
- SQLTable (набор данных Table);
- SQLMonitor (монитор выполнения SQL-запросов);
- SimpleDataSet (клиентский набор данных).

Рис. 14.5. Страница **dbExpress**

Рис. 14.6. Страница **DataSnap**

Страница **DataSnap** (рис. 14.6) содержит компоненты, предназначенные для создания многоуровневых приложений:

- `DCOMConnection` (управление соединением клиентских приложений DCOM с удаленным сервером);
- `SocketConnection` (сокет Windows для управления соединением с сервером приложения);
- `SimpleObjectBroker` (обслуживание списка доступных серверов приложения для компонента соединения);
- `WebConnection` (управление соединением с сервером приложения по протоколу HTTP);
- `ConnectionBroker` (централизация соединения с сервером приложения множества клиентов);
- `SharedConnection` (разделяемое соединение, управляющее соединением с дочерним удаленным модулем данных);
- `LocalConnection` (соединение между клиентским набором данных или посредником XML и провайдером).

Страница **BDE** (рис. 14.7) стала включать на один компонент меньше и содержит компоненты, предназначенные для управления данными с использованием BDE:

- `Table` (набор данных, основанный на таблице БД);
- `Query` (набор данных, основанный на SQL-запросе);
- `StoredProc` (вызов хранимой процедуры сервера);
- `DataBase` (соединение с БД);
- `Session` (текущий сеанс работы с БД);
- `BatchMove` (выполнение операций над группой записей);
- `UpdateSQL` (изменение набора данных, основанного на SQL-запросе или хранимой процедуре);
- `NestedTable` (вложенная таблица).

На странице **ADO** (рис. 14.8) расположены компоненты, предназначенные для управления данными с использованием технологии ADO (Active Data Objects):

- `ADOConnection` (соединение);
- `ADOCommand` (команда);

Рис. 14.7. Страница **BDE**

Рис. 14.8. Страница **ADO**

- `ADODataSet` (набор данных);
- `ADOTable` (набор данных `Table`);
- `ADOQuery` (набор данных `Query`);
- `ADOStoredProc` (вызов хранимой процедуры сервера);
- `RDSConnection` (соединение RDS).

Замечание

Соединение RDS служит для управления передачей объекта `Recordset` от одного процесса (компьютера) к другому при создании серверных приложений.

На странице **InterBase** (рис. 14.9) находятся компоненты, предназначенные для работы с сервером InterBase:

- `IBTable` (набор данных `Table`);
- `IBQuery` (набор данных `Query`);
- `IBStoredProc` (вызов хранимой процедуры);
- `IBDatabase` (соединение с БД);
- `IBTransaction` (транзакция);
- `IBUpdateSQL` (изменение набора данных, основанного на SQL-запросе);
- `IBDataSet` (источник данных);
- `IBSQL` (выполнение SQL-запроса);
- `IBDatabaseInfo` (информация о БД);
- `IBSQLMonitor` (монитор выполнения SQL-запросов);
- `IBEvents` (событие сервера);
- `IBExtract` (извлечение данных);
- `IBClientDataSet` (клиентский источник данных).

Страница **Decision Cube** (рис. 14.10) содержит компоненты, предназначенные для построения систем принятия решений:

- `DecisionCube` (куб многомерных данных);
- `DecisionQuery` (набор, содержащий многомерные данные);
- `DecisionSource` (источник многомерных данных);
- `DecisionPivot` (двумерная проекция многомерных данных);
- `DecisionGrid` (сетка для табличного представления многомерных данных);
- `DecisionGraph` (графическое представление многомерных данных).

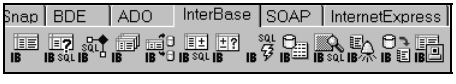


Рис. 14.9. Страница **InterBase**

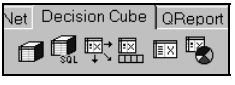


Рис. 14.10. Страница **Decision Cube**

Рис. 14.11. Страница **Rave**

И наконец, на странице **Rave** (рис. 14.11), которая заменила страницу **QReport**, находятся компоненты, предназначенные для построения отчетов:

- `RvProject` (обеспечивает доступ к визуальным отчетам);
- `RvSystem` (обеспечивает просмотр, печать и настройку параметров отчетов);
- `RvNDRWriter` (обеспечивает хранение отчета в двоичном формате);
- `RvCustomConnection` (служит для настройки способа передачи данных в отчет);
- `RvDataSetConnection` (обеспечивает доступ к компонентам, производным от класса `TDataSet`);
- `RvTableConnection` (служит для доступа к компонентам типа `TTable` или их наследникам;
- `RvQueryConnection` (служит для доступа к компонентам типа `TQuery` или их потомкам);
- `RvRenderPreview` (посылает отчет в двоичном формате для предварительного просмотра);
- `RvRenderPrinter` (посылает отчет в двоичном формате на текущее устройство печати);
- `RvRvRenderPDF` (преобразует поток или файл NDR (Rave snapshot report file, файл снимка отчета Rave) в формат PDF);
- `RvRenderHTML` (преобразует поток или файл NDR в формат HTML);
- `RvRenderRTF` (преобразует поток или файл NDR в документ RTF);
- `RvRenderRTF` (преобразует поток или файл NDR в текстовый документ).

Имена многих компонентов, предназначенных для работы с данными, содержат префиксы, например, `DB`, `IB` или `Rv`. Префикс `DB` означает, что визуальный компонент связан с данными и используется для построения интерфейсной части приложения. Такие компоненты размещаются в форме и предназначены для управления данными со стороны пользователя. Префикс `Rv` означает, что компонент используется для построения отчетов Rave. Префикс `IB` означает, что компонент предназначен для работы с сервером InterBase.

Исключения баз данных

В дополнение к рассмотренным ранее исключениям, специально для операций, связанных с БД, система Delphi предоставляет следующие дополнительные классы исключений:

- `EDatabaseError` — ошибка БД; потомки этого класса:
  - `EDBEngineError` — ошибка BDE (для локальных БД и сетевых БД архитектуры "файл-сервер");

- EDBClient — ошибка в приложении клиента (для сетевых БД архитектуры "клиент-сервер"); код ошибки возвращается BDE, ADO, dbExpress или другим механизмом доступа к данным;
- EUpdateError — ошибка, возникающая при обновлении записей;

♦ EDBEditError — введенное в поле значение не соответствует типу поля.

Класс EDatabaseError предназначен для обработки ошибок, возникающих при выполнении операций с набором данных (класс TDataSet и его потомки, в первую очередь, TTable и TQuery). Этот класс производится непосредственно от класса Exception. Исключение класса EDatabaseError генерируется, например, при попытке открыть набор данных, связанный с отсутствующей таблицей, или изменить запись набора данных, который находится в режиме просмотра.

Класс EDBEngineError предназначен для обработки ошибок, возникающих при работе с процессором баз данных BDE на локальном компьютере. В дополнение к свойствам, имеющимся и у класса EDatabaseError, у класса EDBEngineError есть два новых свойства:

♦ ErrorCount типа Integer — содержит количество возникших исключений;

♦ Errors [Index: Integer] типа TDBError — представляет собой список исключений, общее число которых указано свойством ErrorCount. Индекс Index позволяет получить доступ к возникшему исключению по его номеру (нумерация начинается с нуля). На практике обычно анализируется первое из исключений (Errors[0]), указывающее основную причину ошибки.

Класс TDBError содержит информацию об исключении, определяемую следующими свойствами:

♦ Message типа String (текст сообщения, характеризующего возникшую ошибку);

♦ ErrorCode типа DBIResult (код ошибки), тип DBIResult соответствует типу Word;

♦ Category типа Byte (категория исключения);

♦ SubCode типа Byte (группа, или подкод, исключения);

♦ NativeError типа Longint (код ошибки, возвращаемой сервером) — если значение этого свойства равно нулю, то исключение произошло не на сервере.

Код ошибки, определяемый значением свойства ErrorCode, имеет две составляющие: категорию и группу исключения, определяемые значениями свойств Category и SubCode соответственно. В *категорию* объединяются по признаку схожести несколько исключений. *Группа* уточняет исключение внутри категории.

Отметим, что для программной генерации исключений, обслуживаемых классами EDatabaseError и EDBEngineError, используются специальные методы DatabaseError и DBIError, рассматриваемые далее.

Класс EDBClient отличается от класса EDBEngineError в основном тем, что предназначен для обработки ошибок, возникающих при работе с различными механизмами доступа к данным (не только с процессором баз данных BDE) в операциях с сетевыми базами архитектуры "клиент-сервер".

Генерируемые при работе с БД исключения обрабатывают глобальные и локальные обработчики, с которыми мы познакомимся немного позже. Кроме того, используемые для доступа к данным компоненты имеют специальные события для обработки исключений.

Например, для набора данных `Table` такими событиями являются:

- `OnEditError` (ошибка редактирования или вставки записи);
- `OnPostError` и `OnUpdateError` (ошибка закрепления изменений в записи);
- `OnDeleteError` (ошибка удаления записи).

Использование подобных событий и программирование их обработчиков будут рассмотрены при изучении соответствующих компонентов.

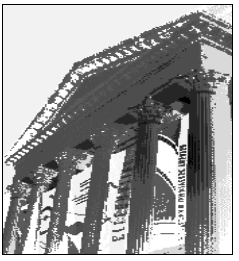
Класс `EDBEditError` используется в случаях, когда вводимые в поле данные несовместимы с маской ввода, заданной с помощью свойства `EditMask` этого поля. Отметим, что для проверки вводимых в поле значений можно также использовать события `OnSetText`, `OnValidate` и `OnChange`. Примеры использования этих событий при программировании обработчиков приводятся в *главе 17*.

Приведем в качестве примера процедуру, в которой проводится анализ исключения, возникшего при работе с базой данных с помощью BDE.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  try
    if OpenDialog1.Execute then begin
      Table1.Active := False;
      Table1.TableName := OpenDialog1.FileName;
      Table1.Active := True;
    end;
  except
    on EO: EDatabaseError do MessageDlg('Ошибка EDatabaseError!',
           mtError, [mbOK], 0);
    on EO: EDBEngineError do MessageDlg('Ошибка EDBEngineError!',
           mtError, [mbOK], 0);
    else MessageDlg('Неопознанная ошибка!', mtError, [mbOK], 0);
  end;
end;
```

При смене таблицы, связанной с набором данных `Table1`, выполняются анализ и обработка возможного исключения. Исключение проверяется на принадлежность к одному из двух классов: `EDatabaseError` и `EDBEngineError`. Обработка заключается в выдаче сообщения о принадлежности исключения к одному из этих классов. Если исключение носит другой характер, то выдается сообщение о том, что оно не распознано.

ГЛАВА 15



Проектирование баз данных

Проектирование реляционной БД заключается главным образом в разработке структуры данных, т. е. в определении состава таблиц и связей между ними. При этом структура должна быть эффективной и обеспечивать:

- быстрый доступ к данным;
- отсутствие дублирования (повторения) данных;
- целостность данных.

Проектирование структуры данных (структуры БД) также называют *логическим* проектированием, или проектированием на логическом уровне.

При проектировании структур данных можно выделить три основных подхода.

- Сбор информации об объектах решаемой задачи в рамках одной таблицы (одного отношения) и последующее разбиение ее на несколько взаимосвязанных таблиц на основе нормализации отношений.
- Формулирование знаний о системе (определение типов исходных данных и их взаимосвязей) и требований к обработке данных, а затем получение с помощью средств CASE схемы БД или готовой прикладной информационной системы.
- Структурирование информации в результате системного анализа на основе совокупности правил и рекомендаций.

Проектирование может выполняться классическим способом, когда разработчик собирает и выделяет объекты системы и их характеристики, после чего вручную приводит их к требуемой структуре данных. Кроме того, для проектирования можно использовать так называемые CASE-системы, которые автоматизируют процесс разработки не только БД, но и информационной системы в целом.

Нормализация базы данных

Нормализация БД — это процесс уменьшения избыточности информации в БД. Метод нормализации основан на достаточно сложной теории реляционных моделей данных. Рассмотрим основные особенности и технику нормализации, не вдаваясь в теоретическое обоснование этих вопросов.

Избыточность данных и аномалии

При разработке структуры БД могут возникнуть проблемы, связанные:

- с избыточностью данных;
- с аномалиями.

Под *избыточностью данных* понимают дублирование данных, содержащихся в БД. При этом различают простое (неизбыточное) дублирование и избыточное дублирование данных.

Избыточность данных при выполнении операций с ними может приводить к различным аномалиям — нарушению целостности БД. Выделяют аномалии:

- удаления;
- обновления;
- ввода.

Неизбыточное дублирование является естественным и допустимым, его примером является список телефонов (местных) сотрудников организации, показанный в табл. 15.1.

Таблица 15.1. *Пример неизбыточного дублирования данных*

Сотрудник	Телефон	Сотрудник	Телефон
Иванов П. Л.	123	Кузнецова В. А.	789
Петров А. Ф.	123	Васин И. Г.	123
Сидоров О. Е.	456		

Три сотрудника имеют одинаковый номер телефона 123, что возможно, например, в случае, когда они находятся в одной комнате. Таким образом, номер телефона в таблице дублируется, однако для каждого сотрудника этот номер является уникальным. В случае удаления одного из дублированных значений номера телефона (удаления соответствующей строки таблицы) будет потеряна информация о сотруднике — Иванове П. Л., Петрове А. Ф. или Васине И. Г., — что является *аномалией удаления*.

При смене номера телефона в комнате его необходимо изменить для всех сотрудников, которые в ней находятся. Если для какого-либо сотрудника этого не сделать, например, для Петрова А. Ф., то возникает несоответствие данных, связанное с *аномалией обновления*.

Аномалия ввода заключается в том, что при вводе в таблицу новой строки для ее полей могут быть введены недопустимые значения. Например, значение не входит в заданный диапазон или не задано значение поля, которое в обязательном порядке должно быть заполнено (не может быть пустым).

Теперь приведем пример *избыточного дублирования* данных. Список телефонов дополнен номерами комнат, в которых находятся сотрудники (табл. 15.2). При этом номер телефона указан только для одного из сотрудников — в примере для Иванова П. Л., который находится в списке первым. Вместо номеров телефонов других сотрудников этой комнаты поставлен прочерк. На практике кодировка прочерка зависит

от особенностей конкретной таблицы, например, можно обозначать прочерк значением `Null`.

Таблица 15.2. *Пример избыточного дублирования данных*

Сотрудник	Комната	Телефон
Иванов П. Л.	17	123
Петров А. Ф.	17	—
Сидоров О. Е.	22	456
Кузнецова В. А.	8	789
Васин И. Г.	17	—

Однако при таком построении таблицы появляются проблемы.

- Номер телефона произвольного сотрудника можно получить только путем поиска в другом столбце таблицы (по номеру комнаты).
- При запоминании таблицы для каждой строки отводится одинаковая память вне зависимости от наличия или отсутствия прочерков.
- При удалении строки с данными сотрудника, для которого указан номер телефона комнаты, будет утеряна информация о номере телефона для всех сотрудников из этой комнаты.

Если вместо прочерков указать номер телефона, то избыточное дублирование все равно остается. От него можно избавиться, например, с помощью разбиения таблицы на две новых (табл. 15.3 и 15.4). Разбиение — это процесс деления таблицы на несколько таблиц с целью поддержания целостности данных, т. е. устранения избыточности данных и аномалий. Таблицы связаны между собой по номеру комнаты. Для получения информации о номере телефона сотрудника из первой таблицы по фамилии сотрудника нужно считать номер его комнаты, после чего из второй таблицы по номеру комнаты считывается номер телефона.

Рассмотренное разбиение таблицы на две является примером нормализации отношений. При этом избыточность данных уменьшилась, однако одновременно увеличилось *время доступа* к ним. Для получения номера телефона сотрудника теперь необходимо работать с двумя таблицами, а не с одной, как в предыдущем случае.

Таблица 15.3. *Список сотрудников и номеров комнат*

Сотрудник	Комната
Иванов П. Л.	17
Петров А. Ф.	17
Сидоров О. Е.	22
Кузнецова В. А.	8
Васин И .Г.	17

Таблица 15.4. *Список номеров комнат и телефонов*

Комната	Телефон
17	123
8	789
22	456

Приведение к нормальным формам

Процесс проектирования БД с использованием метода нормальных форм является итерационным (пошаговым) и заключается в последовательном переводе по определенным правилам отношений из первой нормальной формы в нормальные формы более высокого порядка. Каждая следующая нормальная форма ограничивает определенный тип функциональных зависимостей, устраняет соответствующие аномалии при выполнении операций над отношениями БД и сохраняет свойства предшествующих нормальных форм.

Выделяют такую последовательность нормальных форм:

- первая нормальная форма;
- вторая нормальная форма;
- третья нормальная форма;
- усиленная третья нормальная форма, или нормальная форма Бойса — Кодда;
- четвертая нормальная форма;
- пятая нормальная форма.

Проектирование начинается с определения всех объектов, информация о которых должна содержаться в БД, и выделения атрибутов (характеристик) этих объектов. Атрибуты всех объектов сводятся в одну таблицу, которая является исходной. Затем эта таблица последовательно приводится к нормальным формам в соответствии с их требованиями. На практике обычно используются первые три нормальные формы.

Для примера спроектируем БД для хранения информации о футбольном чемпионате страны. Будем запоминать информацию о дате матча, игравших командах и забитых голах. Сначала объединим все данные в одну исходную таблицу, имеющую следующую структуру (поля):

- дата матча;
- команда хозяев;
- команда гостей;
- игрок, забивший гол;
- признак команды;
- время гола.

В качестве данных о команде (хозяев и гостей) укажем ее название, город и фамилию тренера, что однозначно идентифицирует любую команду. Для игрока, забившего гол, будем указывать фамилию, а для обозначения его принадлежности к той или иной команде используем признак, например, `х` — для команды хозяев, а `г` — для команды гостей. Время гола представляет число минут, прошедших с начала матча.

Приведенная таблица имеет относительно простую структуру, на практике подобные таблицы содержат также данные о составах команд, арбитрах, времени начала игры, числе зрителей, нарушениях правил, заменах, предупреждениях и удалениях игроков и другую информацию. В структуре таблицы указаны только названия полей, поскольку тип и размерность полей на этом этапе большого значения не имеют.

Созданную таблицу можно рассматривать как однотабличную БД. Ее главным недостатком является значительная избыточность данных. Так, для каждого игрока, забившего гол, указываются дата матча и информация о команде хозяев и гостей. Дублирование данных способно привести к аномалиям, а также к существенному увеличению размера базы.

Отметим, что в таблице отсутствует информация о счете матча, т. к. ее можно получить, подсчитав голы, забитые хозяевами и гостями. Побочным явлением такого подхода является то, что если в матче не было забитых голов, то информация о соответствующем матче в таблице отсутствует. Эта ситуация исправляется автоматически при последующем разбиении исходной таблицы на таблицы матчей и голов.

Первая нормальная форма

Приведем исходную таблицу к первой нормальной форме, для которой должны выполняться следующие условия:

- поля содержат неделимую информацию;
- в таблице отсутствуют повторяющиеся группы полей.

Во втором и третьем полях исходной таблицы содержится информация о названии команды, городе и тренере, например, `Зенит, Санкт-Петербург, Дик Адвокаат`. Это противоречит первому требованию — информация о команде, городе и тренере является делимой. Эта информация может и должна быть разделена на три отдельных поля — название команды, город, фамилия тренера.

Согласно второму требованию, в таблице должны отсутствовать повторяющиеся группы полей, т. е. группы, содержащие одинаковые функциональные значения. В исходной таблице таких групп нет, поэтому второму требованию она удовлетворяет. Может показаться, что повторяющимися группами полей являются поля с информацией о командах хозяев и гостей. Несмотря на то, что состав и определения названных полей полностью совпадают, эти поля имеют различное функциональное назначение и не считаются повторяющимися.

Примером таблицы, имеющей повторяющиеся группы полей, может служить табл. 15.5 с результатами экзаменационной сессии. В этой таблице для оценок, полученных студентами на экзаменах, необходимо создать столько полей, сколько может быть различных предметов. В связи с тем, что студент сдает не все экзамены (для них проставлены прочерки), размер записей и таблицы в целом неоправданно увеличивается. Кроме того, значительные трудности создает изменение состава предметов. Если организовать переименование предмета достаточно просто, то его удаление или добавление требует изменения структуры таблицы, что, вообще говоря, крайне нежелательно, поскольку структура таблицы обычно определяется еще на этапе проектирования БД.

Таблица 15.5. *Результаты экзаменационной сессии*

Студент	Математика	Информатика	Физика	История	Английский язык
Семенов Р. О.	4	4	4	—	4
Костина В. К.	4	3	3	—	4
Папаев Д. Г.	5	5	5	—	—
Симонов П. С.	—	—	—	4	4

Вторая и третья нормальные формы касаются отношений между ключевыми и неключевыми полями.

Вторая нормальная форма

Ко *второй нормальной форме* предъявляются следующие требования:

- таблица должна удовлетворять требованиям первой нормальной формы;
- любое неключевое поле должно однозначно идентифицироваться ключевыми полями.

Записи таблицы, приведенной к первой нормальной форме, не являются уникальными и содержат дублированные данные. Так, если за одну минуту футболист забил несколько голов, то таблица будет содержать одинаковые записи. Чтобы обеспечить уникальность записей, введем в таблицу поле ключа — код матча. При этом значение ключа будет однозначно определять каждую запись таблицы.

Тогда структура таблицы примет такой вид:

- Код матча (уникальный ключ)
- Дата матча
- Название команды хозяев
- Город команды хозяев
- Фамилия тренера команды хозяев
- Название команды гостей
- Город команды гостей
- Фамилия тренера команды гостей
- Игрок
- Признак команды
- Время гола

Записи этой таблицы имеют значительное избыточное дублирование данных, т. к. дата матча, а также информация о командах указываются для каждого гола. Поэтому разобьем таблицу на две таблицы, одна из которых будет содержать данные о матчах, а вторая — о голах, забитых в каждом конкретном матче. Структуры этих таблиц будут следующими.

Поля таблицы матчей:

- Код матча (уникальный ключ)
- Дата матча
- Название команды хозяев
- Город команды хозяев
- Фамилия тренера команды хозяев
- Название команды гостей

- Город команды гостей
- Фамилия тренера команды гостей

Поля таблицы голов:

- Код гола (уникальный ключ)
- Код матча
- Игрок
- Признак команды
- Время гола

Таблицы связаны по полю "Код матча", которое для таблицы матча имеет уникальное значение. Чтобы обеспечить уникальность записей таблицы голов, в нее введено ключевое поле "Код гола".

Третья нормальная форма

Требования *третьей нормальной формы*:

- таблица должна удовлетворять требованиям второй нормальной формы;
- ни одно из неключевых полей не должно однозначно идентифицироваться значением другого неключевого поля (полей).

Приведение таблицы к третьей нормальной форме предполагает выделение в отдельную таблицу (таблицы) тех полей, которые не зависят от ключа. В таблице матчей такими являются поля с фамилиями тренеров команд, которые однозначно определяются значениями названия и города команды. (Предполагается, что в течение сезона у команды тренер не меняется, на практике это часто не выполняется, но не относится к сути рассматриваемого вопроса.) Разобьем таблицу матчей на две таблицы: одну с данными о матчах и вторую — с данными о командах. Их структура будет такой:

Таблица матчей:

- Код матча (уникальный ключ)
- Дата матча
- Код команды хозяев
- Код команды гостей

Таблица команд:

- Код команды (уникальный ключ)
- Название команды
- Город команды
- Фамилия тренера команды

Информация о команде хранится в строковых полях, которые имеют достаточно большой размер — в нашем случае приблизительно 60 символов. На практике о каждой команде необходимо запоминать больше данных, чем просто ее название, город приписки и фамилия тренера, что существенно увеличивает объем информации. Поэтому

для уменьшения размера записей таблицы матчей не только фамилия тренера, но и вся подробная информация о командах вынесена в таблицу команд. Вместо этого команды хозяев и гостей в таблице матчей идентифицируются кодом команды, который является уникальным ключевым значением в таблице команд.

После приведения к третьей нормальной форме база данных с информацией о футбольном чемпионате страны будет иметь структуру, показанную на рис. 15.1, где кроме описания таблиц обозначены связи между ними.

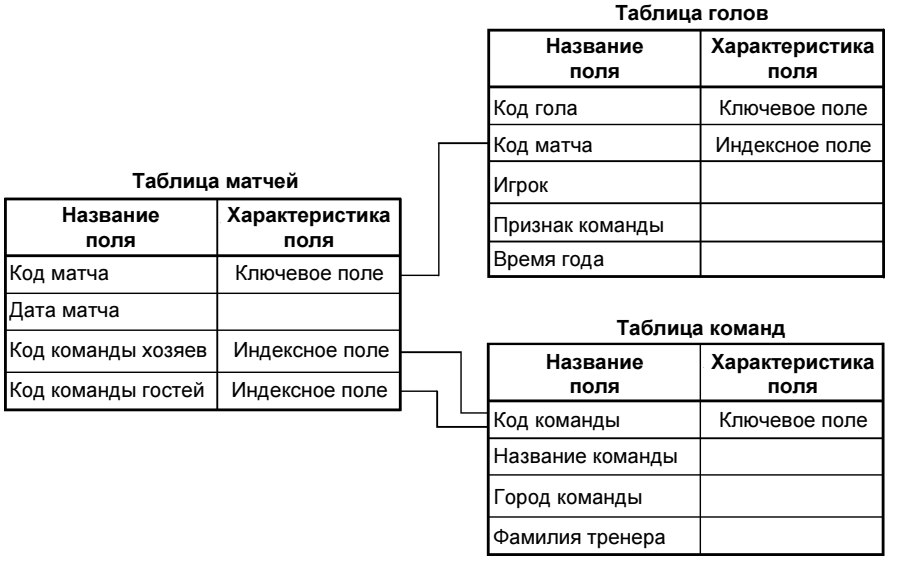


Рис. 15.1. Структура БД "Чемпионат по футболу"

Следование требованиям нормализованных форм не всегда является обязательным. Как уже отмечалось, с ростом числа таблиц структура БД усложняется и возрастает время доступа к данным. В ряде случаев для упрощения структуры БД можно позволить частичное дублирование данных, не допуская, однако, нарушения их целостности и сохраняя их непротиворечивость.

В рассмотренной БД это относится, например, к информации о счете матча. Чтобы исключить дублирование данных, для него не отведено отдельное поле. Узнать счет матча можно с помощью двух запросов к БД, возвращающих число мячей, забитых в указанном матче хозяевами и гостями соответственно. Запрос может иметь следующий вид:

```
SELECT COUNT(G_OwnerSign)
    FROM Goal
    WHERE Goal.G_Match = :MatchCode
      AND Goal.G_OwnerSign = :OwnerSign
```

Номер матча и признак команды, игрок которой забил гол, передаются в запрос через параметры `MatchCode` и `OwnerSign`, перед которыми указывается двоеточие *(см. разд. "Использование хранимых процедур" в главе 26)*.

Поскольку процесс обработки данных усложнился, а время доступа к ним возросло, возможным вариантом для структуры рассмотренной БД является такой, когда счет матча запоминается в отдельном поле таблицы матчей. При этом необходимо обеспечить целостность данных при изменении БД с учетом того факта, что часть ее данных дублирована.

Средства CASE

В предыдущем разделе проектирование БД выполнено вручную. Разработчик сам осуществляет такие операции, как определение состава полей, распределение их по таблицам, а также установление связей между таблицами. Ручное проектирование применяется для разработки БД самого различного назначения и для относительно небольших БД вполне приемлемо. Однако с ростом размера базы данных, когда в нее включаются от нескольких десятков до сотен различных таблиц, возникает проблема сложности организации данных, в том числе установления взаимосвязей между таблицами. Для облегчения решения этой проблемы предназначены системы автоматизации разработки приложений, или средства CASE (Computer Aided Software Engineering).

Средства CASE представляют собой программы, поддерживающие процессы создания и/или сопровождения информационных систем, такие как анализ и формулировка требований, проектирование БД и приложений, генерация кода, тестирование, обеспечение качества, управление конфигурацией и проектом. То есть средства CASE позволяют решать более масштабные задачи, чем просто проектирование БД. Кстати, согласно предлагаемой далее классификации, система Delphi также относится к типу CASE, т. к. позволяет автоматизировать разработку приложений.

Систему CASE можно определить как набор средств CASE, имеющих определенное функциональное предназначение и выполненных в рамках единого программного продукта.

Классификация средств (систем) CASE, используемых для разработки баз данных, производится по следующим признакам:

- ориентация на этапы жизненного цикла;
- функциональная полнота;
- тип используемых моделей;
- степень независимости от СУБД;
- платформа.

По *ориентации на этапы жизненного цикла* можно выделить следующие основные типы систем CASE (в скобках приведены названия фирм-разработчиков):

- системы анализа, предназначенные для построения и анализа моделей предметной области, например, Design/IDEF (Meta Software) и BPWin (Logic Works);
- системы анализа и проектирования, поддерживающие и обеспечивающие создание проектных спецификаций, например, Vantage Team Builder (Cayenne), Silverrun (Silverrun Technologies), PRO-I (McDonnell Douglas);

- системы проектирования БД, обеспечивающие моделирование данных и разработку схем баз данных для основных СУБД, например, ERwin (Logic Works), SDesigner (SPD), DataBase Designer (Oracle);
- системы разработки приложений, например, Uniface (Compuware), JAM (JYACC), PowerBuilder (Sybase), Developer/2000 (Oracle), New Era (Informix), SQL Windows (Centura), Delphi (Borland).

По *функциональной полноте* системы CASE условно делятся на следующие группы:

- системы, предназначенные для решения частных задач на одном или нескольких этапах жизненного цикла, например, ERwin (Logic Works), S-Designer (SPD), CASE.Аналитик (МакроПроджект) и Silverrun (Silverrun Technologies);
- интегрированные системы, поддерживающие весь жизненный цикл информационной системы и связанные с общим репозиторием (хранилищем), например, система Vantage Team Builder (Cayenne) и система Designer/2000 с системой разработки приложений Developer/2000 (Oracle).

По *типу используемых моделей* системы CASE делятся на три разновидности: структурные, объектно-ориентированные и комбинированные.

Исторически первыми появились структурные системы CASE, которые основываются на методах структурного и модульного программирования, структурного анализа и синтеза, например, Vantage Team Builder (Cayenne).

Объектно-ориентированные системы CASE получили массовое распространение с начала 90-х годов XX века. Они позволяют сократить сроки разработки, а также повысить надежность и эффективность функционирования информационной системы. Примерами объектно-ориентированных систем CASE являются Rational Rose (Rational Software) и Object Team (Cayenne).

Комбинированные системы CASE поддерживают одновременно и структурное, и объектно-ориентированное программирование, например, Designer/2000 (Oracle).

По *степени независимости от СУБД* системы CASE делятся на две группы:

- независимые системы;
- встроенные в СУБД системы.

Независимые системы CASE поставляются в виде автономных систем, не входящих в состав конкретной СУБД. Обычно они поддерживают несколько форматов баз данных через интерфейс ODBC. К числу независимых относятся SDesigner (SDP, Powersoft), ERwin (LogicWorks), Silverrun (Silverrun Technologies).

Встроенные системы CASE обычно поддерживают главным образом формат базы данных, в состав системы управления которой они входят. При этом возможна поддержка форматов и других баз данных. Примером встроенной системы является система Designer/2000, входящая в состав СУБД Oracle.

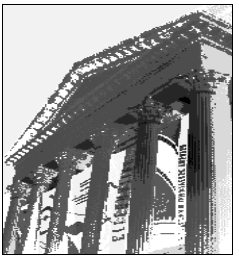
Платформа определяет компьютер и операционную систему, на которых допускается использовать продукт, созданный с помощью данной системы CASE.

Перечислим средства CASE, которые можно применять при разработке БД и приложений с помощью Delphi.

- ModelMaker — продукт, поставляемый вместе с Delphi 7. Служит для разработки классов и пакетов компонентов для Delphi. Представляет собой CASE-средство, ориентированное на генерацию кода Delphi. Позволяет хранить и обслуживать отношения между классами и их членами, поддерживает построение UML-диаграмм. По сравнению с другими генераторами кода ModelMaker позволяет разрабатывать сложные проекты.
- Data Module Designer — позволяет проектировать БД с таблицами формата Paradox. Программа обеспечивает достаточно удобный и наглядный интерфейс. Структура БД, в том числе связи между таблицами, отображается в графическом виде.
- Cadet — независимый продукт, позволяющий проектировать БД с таблицами форматов dBase, Paradox и InterBase. С учетом того, что указанные форматы являются родными для Delphi, программу Cadet удобно использовать при разработке информационных систем.

Программы Data Module Designer и Cadet предназначены для моделирования структур данных и автоматизации проектирования БД. Возможности, предоставляемые этими средствами, меньше, чем возможности таких мощных систем, как, например, Sdesigner, однако доступность делает их привлекательными для использования. Так, программа Cadet является условно-бесплатной, а Data Module Designer входит в состав СУБД Paradox 7.0. Впрочем, с появлением ModelMaker использование других CASE-средств может не потребоваться.

ГЛАВА 16



Технология создания информационной системы

Продемонстрируем возможности Delphi по работе с БД на примере создания простой информационной системы. Эту информационную систему можно разработать даже без написания кода: все необходимые операции выполняются с помощью программы Database Desktop, Конструктора формы и Инспектора объектов. Работа над информационной системой состоит из следующих основных этапов:

- создание БД;
- создание приложения.

Кроме приложения и БД, в информационную систему также входят вычислительная система и СУБД. Предположим, что компьютер или компьютерная сеть уже существуют, и их характеристики удовлетворяют потребностям будущей информационной системы. В качестве СУБД выберем Delphi.

В простейшем случае БД состоит из одной таблицы. Если таблицы уже имеются, то первый этап не выполняется. Отметим, что совместно с Delphi поставляется большое количество примеров приложений, в том числе и приложений БД. Файлы таблиц для этих приложений находятся в каталоге c:\Program Files\Common Files\Borland Shared\Data. Готовые таблицы также можно использовать для своих приложений.

Создание таблиц базы данных

Для работы с таблицами БД при проектировании приложения удобно использовать программу Database Desktop, которая позволяет:

- создавать таблицы;
- изменять структуры;
- редактировать записи.

Кроме того, с помощью Database Desktop можно выполнять и другие действия над БД (создание, редактирование и выполнение визуальных и SQL-запросов, операции с псевдонимами), которые будут рассматриваться в *главе 24*, посвященной инструментам.

Замечание

Почти все рассматриваемые далее действия по управлению структурой таблицы можно выполнить также программно, что описано в *главах 19* и *20*, посвященных навигационному и реляционному способам доступа.

Процесс создания новой таблицы начинается с вызова команды **File | New | Table** (Файл | Новая | Таблица) и происходит в интерактивном режиме. При этом разработчик должен:

- выбрать формат (тип) таблицы;
- задать структуру таблицы.

В начале создания новой таблицы в окне **Create Table** (Создание таблицы) (рис. 16.1) выбирается ее формат. По умолчанию предлагается формат таблицы Paradox версии 7, который мы и будем использовать. Для таблиц других форматов, например dBase IV, действия по созданию таблицы практически не отличаются.

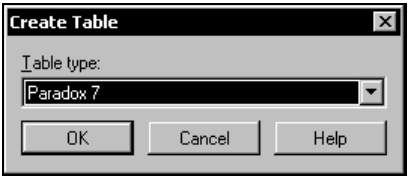


Рис. 16.1. Выбор формата таблицы

После выбора формата таблицы появляется окно определения структуры таблицы (рис. 16.2), в котором выполняются следующие действия:

- описание полей;
- задание ключа;
- задание индексов;
- определение ограничений на значения полей;
- определение условий (ограничений) ссылочной целостности;
- задание паролей;
- задание языкового драйвера;
- задание таблицы для выбора значений.

В этом списке обязательным является только первое действие, т. е. каждая таблица должна иметь хотя бы одно поле. Остальные действия выполняются при необходимости. Часть действий, такие как задание ключа и паролей, производится только для таблиц определенных форматов, например, для таблиц Paradox.

При создании новой таблицы, сразу после выбора ее формата можно не создавать структуру таблицы, а скопировать ее из другой таблицы: при нажатии кнопки **Borrow** (Взаймы) открывается окно **Select Borrow Table** (Выбор таблицы для заимствования) (рис. 16.3).

В этом окне можно выбрать таблицу (ее главный файл) и указать копируемые элементы структуры, установив соответствующий флажок, например, **Primary index** (Пер-

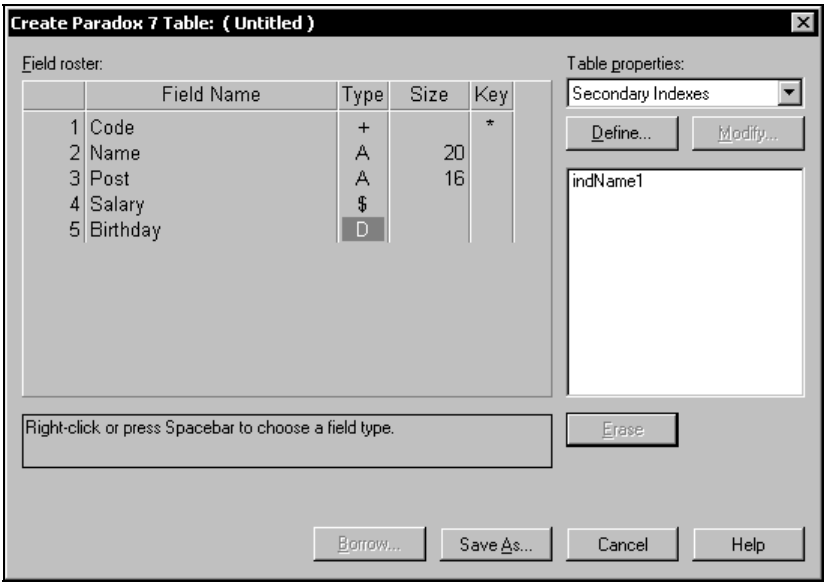


Рис. 16.2. Определение структуры таблицы

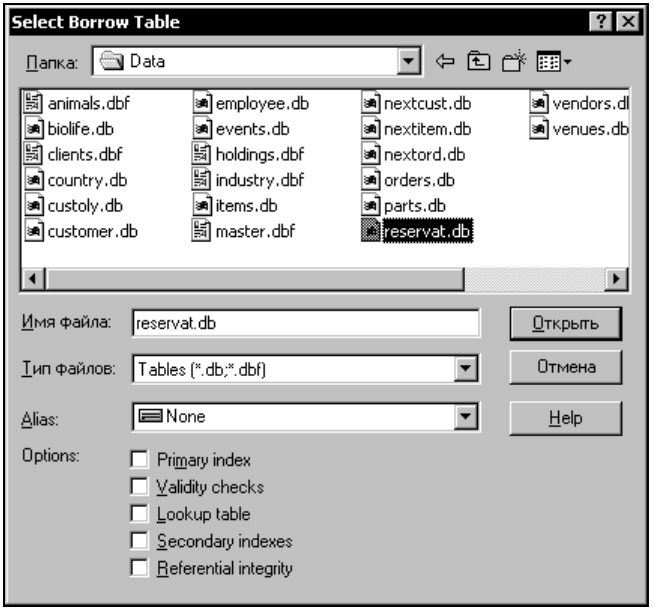


Рис. 16.3. Выбор таблицы для заимствования ее структуры

вичный индекс) для ключа. После нажатия кнопки **Открыть** из выбранной таблицы в новую копируются описания полей, а также те элементы, для которых установлен флажок. Если какой-либо элемент в структуре копируемой таблицы отсутствует, то состояние флажка не имеет значения. Например, если в выбранной таблице не опреде-

лены ограничения ссылочной целостности, то в новой таблице они не появятся, даже если установлен флажок **Referential integrity** (Ссылочная целостность).

Впоследствии скопированную структуру можно настраивать, изменяя, добавляя или удаляя ее отдельные элементы.

После определения структуры таблицы ее необходимо сохранить, нажав кнопку **Save As** и указав расположение таблицы на диске и ее имя. В результате на диск записывается новая таблица, первоначально пустая, при этом все необходимые файлы создаются автоматически.

Описание полей

Центральной частью окна определения структуры таблицы является список **Field roster** (Список полей), в котором указываются поля таблицы. Для каждого поля задаются:

- имя поля — в столбце **Field Name**;
- тип поля — в столбце **Type**;
- размер поля — в столбце **Size**.

Имя поля вводится по правилам, установленным для выбранного формата таблиц. Правила именования и допустимые типы полей таблиц Paradox описаны в *главе 14*.

Тип поля можно задать, непосредственно указав соответствующий символ, например, `A` для символьного или `I` для целочисленного поля, или выбрать его в списке (рис. 16.4), раскрываемом нажатием клавиши <Пробел> или щелчком правой кнопки мыши в столбце **Type**. Список содержит все типы полей, допустимые для заданного формата таблицы. В списке подчеркнуты символы, используемые для обозначения соответствующего типа, при выборе типа эти символы автоматически заносятся в столбец **Type**.

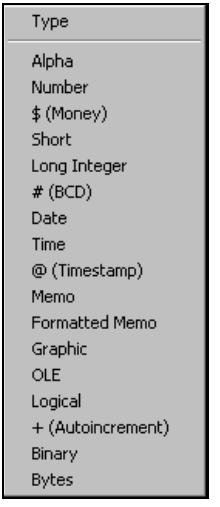


Рис. 16.4. Список типов для полей таблицы Paradox 7

Размер поля задается не всегда, необходимость его указания зависит от типа поля. Для полей определенного типа, например, автоинкрементного (+) или целочисленного (I), размер поля не задается. Для поля строкового типа размер определяет максимальное число символов, которые могут храниться в поле.

Добавление к списку полей новой строки выполняется переводом курсора вниз на несуществующую строку, в результате чего эта строка появляется в конце списка. Вставка новой строки между существующими строками с уже описанными полями выполняется нажатием клавиши <Insert>. Новая строка вставляется перед строкой, в которой расположен курсор. Для удаления строки необходимо установить курсор на эту строку и нажать комбинацию клавиш <Ctrl>+<Delete>.

Ключ создается указанием его полей. Для указания ключевых полей в столбце ключа (**Key**) нужно установить символ *, переведя в эту позицию курсор и нажав любую алфавитно-цифровую клавишу. При повторном нажатии клавиши отметка принадлежности поля ключу снимается. В структуре таблицы ключевые поля должны быть первыми, т. е. верхними в списке полей. Часто для ключа используют автоинкрементное поле (см. рис. 16.2).

Напомним, что для таблиц Paradox ключ также называют первичным индексом (Primary Index), а для таблиц dBase ключ не создается, и его роль выполняет один из индексов.

Для выполнения остальных действий по определению структуры таблицы используется комбинированный список **Table properties** (Свойства таблицы) (см. рис. 16.2), содержащий следующие пункты:

- **Secondary Indexes** (вторичные индексы);
- **Validity Checks** (проверка правильности ввода значений полей) — выбирается по умолчанию;
- **Referential Integrity** (ссылочная целостность);
- **Password Security** (пароли);
- **Table Language** (язык таблицы, языковой драйвер);
- **Table Lookup** (таблица выбора);
- **Dependent Tables** (подчиненные таблицы).

После выбора какого-либо пункта этого списка в правой части окна определения структуры таблицы появляются соответствующие элементы, с помощью которых выполняются дальнейшие действия.

Состав данного списка зависит от формата таблицы. Так, для таблицы dBase он содержит только пункты **Indexes** и **Table Language**.

Задание индексов

Задание индекса сводится к определению:

- состава полей;
- параметров;
- имени.

Эти элементы устанавливаются или изменяются при выполнении операций создания, изменения и удаления индекса.

Замечание

Напомним, что для таблиц Paradox индекс называют также вторичным индексом.

Для выполнения операций, связанных с заданием индексов, необходимо выбрать пункт **Secondary Indexes** (Вторичные индексы) списка **Table properties** (Свойства таблицы), при этом под списком появляются кнопки **Define** (Определить) и **Modify** (Изменить), список индексов и кнопка **Erase** (Удалить). В списке индексов выводятся имена созданных индексов, на рис. 16.2 это индекс `indName1`.

Создание нового индекса начинается с нажатия кнопки **Define**, которая всегда доступна. Она открывает окно **Define Secondary Index** (Задание вторичного индекса), в котором задаются состав полей и параметры индекса (рис. 16.5).

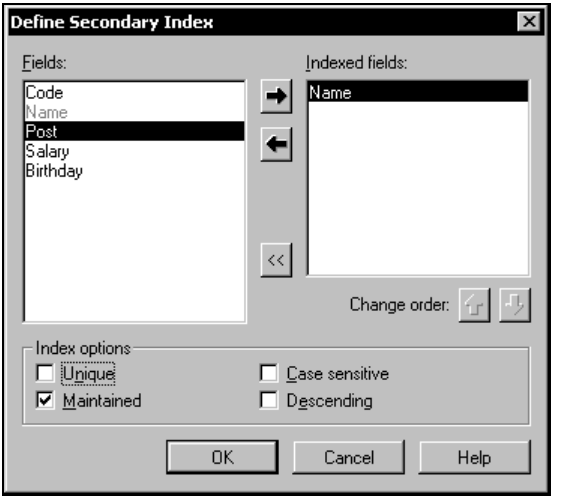


Рис. 16.5. Окно задания индекса

В списке **Fields** окна выводятся имена всех полей таблицы, включая и те, которые нельзя включать в состав индекса, например, графическое поле или поле комментария. В списке **Indexed fields** (Индексные поля) содержатся поля, которые включаются в состав создаваемого индекса. Перемещение полей между списками выполняется выделением нужного поля (полей) и нажатием расположенных между этими списками кнопок с изображением горизонтальных стрелок. Имена полей, которые нельзя включать в состав индекса, выделяются в левом списке серым цветом. Поле не может быть повторно включено в состав индекса, если оно уже выбрано и находится в правом списке.

Замечание

При работе с записями индексные поля обрабатываются в порядке следования этих полей в составе индекса. Это нужно учитывать при указании порядка полей в индексе.

Изменить порядок следования полей в индексе можно с помощью кнопок с изображением вертикальных стрелок, имеющих общее название **Change order** (Изменить порядок). Для перемещения поля (полей) необходимо его (их) выделить и нажать нужную кнопку.

Флажки, расположенные в нижней части окна задания индекса, позволяют указать следующие параметры индекса:

- **Unique** — индекс требует для составляющих его полей уникальных значений;
- **Maintained** — задается автоматическое обслуживание индекса;
- **Case sensitive** — для полей строкового типа учитывается регистр символов;
- **Descending** — сортировка выполняется в порядке убывания значений.

Так как у таблиц dBase нет ключей, для них использование параметра **Unique** является единственной возможностью обеспечить уникальность записей на физическом уровне (уровне организации таблицы), не прибегая к программированию.

После задания состава индексных полей и нажатия кнопки **OK** появляется окно **Save Index As**, в котором нужно указать имя индекса (рис. 16.6). Для удобства обращения к индексу в его имя можно включить имена полей, указав какой-нибудь префикс, например `ind`. Нежелательно образовывать имя индекса только из имен полей, т. к. для таблиц Paradox подобная система именования используется при автоматическом образовании имен для обозначения ссылочной целостности между таблицами. После повторного нажатия кнопки **OK** сформированный индекс добавляется к таблице, и его имя появляется в списке индексов.

Созданный индекс можно изменить, определив новый состав полей, параметров и имени индекса. Изменение индекса практически не отличается от его создания. После выделения индекса в списке и нажатия кнопки **Modify** снова открывается окно задания индекса (см. рис. 16.5). При нажатии кнопки **OK** появляется окно сохранения индекса (см. рис. 16.6), содержащее имя изменяемого индекса, которое можно исправить или оставить прежним.

Для удаления индекса его нужно выделить в списке индексов и нажать кнопку **Erase**. В результате индекс удаляется без предупреждающих сообщений.

Кнопки **Modify** и **Erase** доступны, только если индекс выбран в списке.

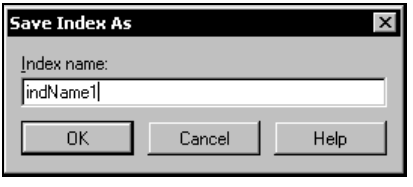


Рис. 16.6. Задание имени индекса

Задание ограничений на значения полей

Задание ограничений на значения полей заключается в указании для полей:

- требования обязательного ввода значения;
- минимального значения;

- максимального значения;
- значения по умолчанию;
- маски ввода.

Замечание

Установленные ограничения задаются на физическом уровне (уровне таблицы) и действуют для любых программ, выполняющих операции с таблицей: как для программ типа Database Desktop, так и для приложений, создаваемых в Delphi. Дополнительно к этим ограничениям или вместо них в приложении можно задать *программные* ограничения.

Для выполнения операций, связанных с заданием ограничений на значения полей, нужно выбрать пункт **Validity Checks** (Проверка значений) комбинированного списка **Table properties** (см. рис. 16.2), при этом ниже списка появляются флажок **Required Field** (Обязательное поле), поля редактирования **Minimum Value**, **Maximum Value**, **Default Value**, **Picture** (Маска ввода) и кнопка **Assist** (Помощь). Флажок и поля редактирования отображают установки для поля таблицы, которое выбрано в списке (курсор находится в строке этого поля).

Требование обязательного ввода значения означает, что поле не может быть пустым (иметь значение `Null`). Это требование действует при добавлении к таблице новой записи. До того как изменения в таблице будут подтверждены, поле должно получить какое-либо непустое значение, в противном случае генерируется ошибка. Ошибка может также возникнуть при редактировании записи, когда будет удалено старое значение поля и не присвоено новое.

Данное требование удобно использовать для так называемых *обязательных полей* таблиц, например, для поля фамилии в таблице сотрудников организации.

Замечание

Обязательность ввода значения не действует на автоинкрементное поле, которое и без того является обязательным и автоматически заполняемым.

Для указания обязательности ввода значения в поле необходимо установить флажок **Required Field**, который по умолчанию снят.

Для полей некоторых типов, в первую очередь числовых, денежных, строковых и даты, иногда удобно задавать диапазон возможных значений, а также значение по умолчанию. Диапазон определяется минимальным и максимальным возможными значениями, которые вводятся в полях редактирования **Minimum Value** и **Maximum Value**. После их задания выход значения поля за указанные границы не допускается при вводе и редактировании любым способом.

Значение поля по умолчанию указывается в поле **Default Value**. Это значение устанавливается при добавлении новой записи, если при этом для поля не указано какое-либо значение.

Замечание

Задание диапазона и значений по умолчанию возможно не для всех полей, например, они не определяются для графического поля и поля комментария. Для этих полей соответствующее поле ввода в диалоговом окне определения структуры таблицы (см. рис. 16.2) блокируется.

В поле ввода **Picture** можно задать маску (шаблон) для ввода значения поля. Ввод по маске поддерживается, например, для таких типов полей, как числовой или строковый. Его удобно использовать для ввода информации определенных форматов, например, телефонных номеров или почтовых индексов.

Для маски используются следующие символы:

- `#` (цифра);
- `?` (любая буква; регистр не учитывается);
- `&` (любая буква; преобразуется к верхнему регистру);
- `~` (любая буква; преобразуется к нижнему регистру);
- `@` (любой символ);
- `!` (любой символ; преобразуется к верхнему регистру);
- `;` (за этим символом следует буква);
- `*` (число повторов следующего символа);
- `[abc]` или `{a,b,c}` (любой из приведенных символов — `a`, `b`, или `c`; во втором случае значения перечисляются через запятую без пробелов).

Маску можно ввести в поле ввода **Picture** вручную или использовать для этого диалоговое окно **Picture Assistance** (Помощник представления), вызываемое нажатием кнопки **Assist** (рис. 16.7).

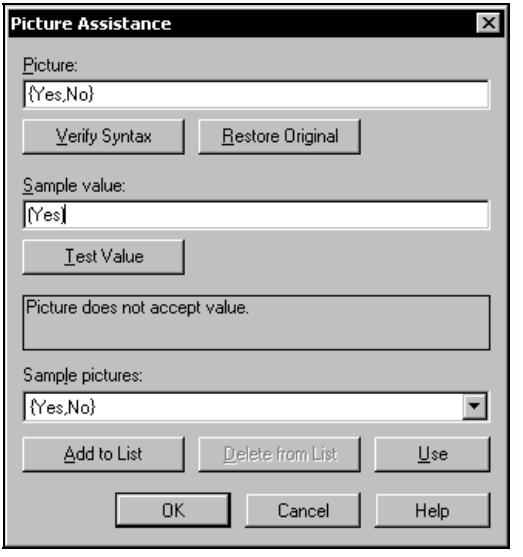


Рис. 16.7. Диалоговое окно формирования маски

Указанное окно помогает ввести, выбрать или откорректировать маску, а также проверить ее функционирование.

Список **Sample pictures** содержит образцы масок, которые выбираются нажатием кнопки **Use**. Выбранная маска помещается в поле ввода **Picture** и доступна для измене-

ния. Для модификации списка образцов масок служат кнопки **Add to List** и **Delete from List**: первая добавляет к списку маску, содержащуюся в поле ввода **Picture**, а вторая удаляет из списка **Sample pictures** выбранную маску.

Проверка синтаксиса маски выполняется по нажатию кнопки **Verify Syntax**, результат проверки выводится в информационной панели. Кнопка **Restore Original** (Вернуть исходную) служит для восстановления начального (т. е. до начала редактирования) значения маски.

Функционирование маски можно проверить, введя в поле ввода **Sample value** значение поля таблицы. По нажатию кнопки **Test Value** выполняется проверка введенного значения, результат проверки выводится в информационной панели.

Задание ссылочной целостности

Понятие ссылочной целостности относится к связанным таблицам и проявляется в следующих вариантах взаимодействия таблиц:

- запрещается изменять поле связи или удалять запись главной таблицы, если для нее имеются записи в подчиненной таблице;
- при удалении записи в главной таблице автоматически удаляются соответствующие ей записи в подчиненной таблице (каскадное удаление).

Для выполнения операций, связанных с заданием ссылочной целостности, необходимо выбрать пункт **Referential Integrity** комбинированного списка **Table properties** (см. рис. 16.2). При этом, как и в случае задания индексов, появляются кнопки **Define**, **Modify**, **Erase** и список, в котором выводятся имена созданных условий ссылочной целостности.

Условие ссылочной целостности задается для подчиненной таблицы и определяется следующими элементами:

- полями связи подчиненной таблицы;
- именем главной таблицы;
- полями связи главной таблицы;
- параметрами.

Разработчик может создать, изменить или удалить условие ссылочной целостности.

Для задания условия ссылочной целостности нужно нажать кнопку **Define**, после чего появляется окно **Referential Integrity** (рис. 16.8).

В списке **Fields** следует выбрать поле связи и нажатием кнопки со стрелкой вправо перевести его в список **Child fields** (Дочерние поля). Если полей связи несколько, то эти действия выполняются для каждого из них. Кнопка со стрелкой влево удаляет выбранное поле из списка дочерних полей.

В списке **Table** указывается главная таблица, имена таблиц выбираются в рабочем каталоге программы Database Desktop (каталог, определенный как Working Directory, его задание описано в *главе 24*). После выбора таблицы и нажатия кнопки со стрелкой влево (рядом со списком таблиц) в поле **Parent's key** автоматически заносятся ключевые поля главной таблицы.

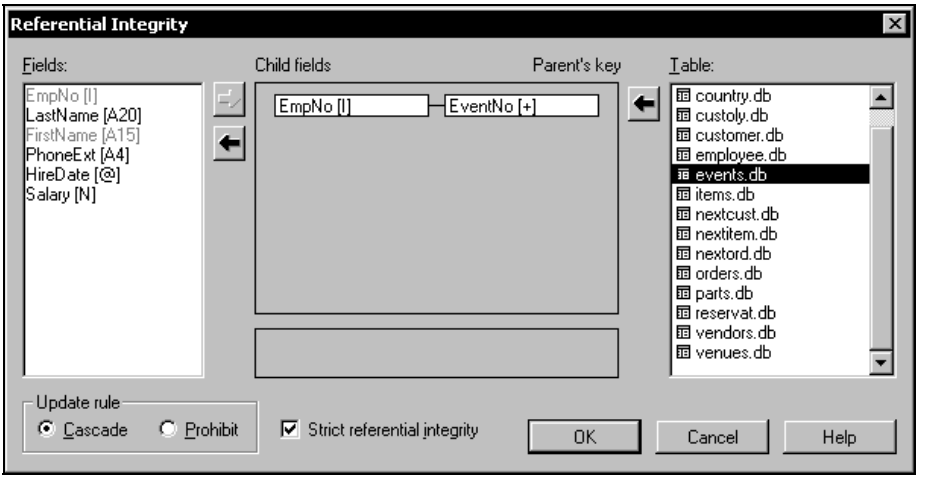


Рис. 16.8. Задание условия ссылочной целостности

Замечание

Главная таблица должна быть закрыта и не использоваться другими программами, включая Database Desktop. В противном случае при попытке сохранить структуру таблицы возникает ошибка.

Параметры ссылочной целостности выбираются переключателями. Группа **Update rule** (Правила изменения) определяет вид взаимодействия таблиц при изменениях в главной таблице. Переключатель **Cascade** устанавливает режим каскадного удаления записей в подчиненной таблице при удалении соответствующей записи главной таблицы. Переключатель **Prohibit** устанавливает режим запрещения изменения поля связи или удаления записи главной таблицы, для которой имеются записи в подчиненной таблице.

Флажок **Strict referential integrity** (Жесткая ссылочная целостность) устанавливает защиту таблиц от модификации с использованием ранних версий программы Database Desktop (под DOS), которые не поддерживают ссылочную целостность.

После установки нужных режимов и нажатия кнопки **OK** появляется окно **Save Referential Integrity As**, в котором нужно указать имя условия (рис. 16.9). Напомним, что для таблиц Paradox условия ссылочной целостности именуются. Для удобства обращения к условию в его имя можно включить имена полей и таблиц, задав при этом некоторый префикс, например, `ri`. После нажатия кнопки **OK** сформированное условие ссылочной целостности добавляется к таблице, и его имя появляется в списке условий.

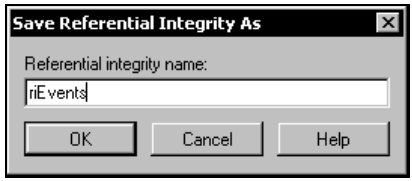


Рис. 16.9. Задание имени для условия ссылочной целостности

Замечание

Условия ссылочной целостности задаются на физическом уровне и действуют для любых программ, выполняющих операции с таблицей: как для программ типа Database Desktop, так и для приложений, создаваемых в Delphi.

Созданное условие ссылочной целостности можно изменить, определив новый состав полей и новые значения параметров. Изменение условия ссылочной целостности практически не отличается от его создания: после выделения имени условия в списке и нажатия кнопки **Modify** открывается окно определения ссылочной целостности (см. рис. 16.8). При нажатии кнопки **OK** измененное условие ссылочной целостности сохраняется под тем же именем.

Для удаления условия ссылочной целостности нужно выделить его в списке и нажать кнопку **Erase**. Удаление производится без выдачи предупреждающих сообщений.

Кнопки **Modify** и **Erase** доступны, только если выбрано условие в списке.

Задание паролей

Пароль позволяет задать права доступа пользователей (приложений) к таблице. Если для таблицы установлен пароль, то он будет автоматически запрашиваться при каждой попытке открытия таблицы.

Замечание

Пароль действует на физическом уровне и его действие распространяется на все программы, выполняющие доступ к таблице: как на программы типа Database Desktop, так и на создаваемые приложения Delphi.

Для выполнения операций, связанных с заданием пароля, нужно выбрать строку **Password Security** в комбинированном списке **Table properties** окна определения структуры таблицы (см. рис. 16.2). При этом под списком становятся доступными кнопки **Define** и **Modify**. Нажатие кнопки **Define** открывает окно **Password Security** (рис. 16.10), в котором задается пароль.

Рис. 16.10. Задание главного пароля

Пароль таблицы вводится дважды — в полях **Master password** (Главный пароль) и **Verify master password** (Подтвердить главный пароль). При нажатии кнопки **OK** оба

значения сверяются, и при их совпадении пароль принимается. Когда пароль определен, кнопка **Define** блокируется и становится доступной кнопка **Modify** изменения пароля. Ее нажатие снова вызывает окно задания пароля (см. рис. 16.10), в котором появляются кнопки **Change** и **Delete**, а поля ввода заблокированы.

Нажатие кнопки **Delete** удаляет пароль, после чего его можно ввести заново. Если в качестве значения пароля указана пустая строка, то пароль для таблицы не задан. Нажатием кнопки **Change** поля ввода разблокируются, и значение пароля можно изменить (это нужно сделать в обоих полях). При этом название кнопки **Change** изменяется на **Revert** (Возврат), и ее повторное нажатие возвращает значение пароля, которое было до начала редактирования.

Рассмотренный пароль считается главным паролем, который предоставляет пользователю *полные права доступа* к таблице, включая изменение записей и структуры таблицы, в том числе смену пароля. Кроме главного пароля, можно задать для таблицы дополнительные пароли, устанавливающие пользователю *ограниченные права доступа* к таблице. Для задания дополнительных паролей нажатием кнопки **Auxiliary Passwords** открывается одноименное окно (рис. 16.11).

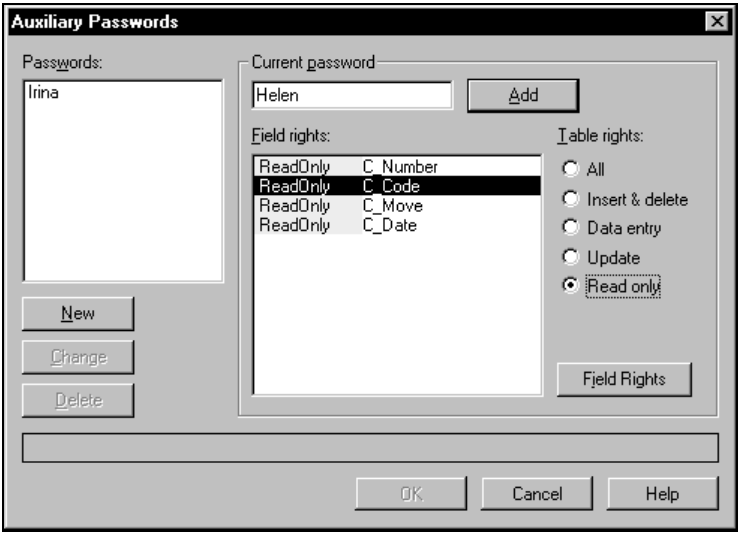


Рис. 16.11. Задание дополнительных паролей

В списке **Passwords** выводятся действующие дополнительные пароли. Группа переключателей **Table rights** определяет для пароля права доступа к таблице в целом. Они могут быть следующими:

- **All** — полные права, включая изменение записей и структуры таблицы;
- **Insert & delete** — разрешены вставка и удаление, а также редактирование записей, запрещено изменение структуры таблицы;
- **Data entry** — разрешены редактирование и вставка записей, запрещены изменение структуры таблицы и удаление записей;

- **Update** — разрешены только просмотр (чтение) записей и редактирование неключевых полей;
- **Read only** — разрешен только просмотр (чтение) записей.

Права доступа к таблице действуют на все ее поля, кроме того, для каждого поля можно установить отдельные права доступа, не зависящие от прав доступа к другим полям. Права доступа к полям выводятся слева от имени поля в списке **Fields rights** и могут иметь следующие значения:

- **All** (чтение и изменение значения поля);
- **ReadOnly** (только чтение значения поля);
- **None** (доступ к полю запрещен).

Смена права доступа к полю выполняется выбором поля в списке и нажатием кнопки **Fields Rights**, при котором право циклически устанавливается очередным значением списка (**All**, **ReadOnly** и **None**).

Создание пароля начинается нажатием кнопки **New**, после чего его имя указывается в поле **Current password** (Текущий пароль) и устанавливаются права доступа к таблице и ее полям. Нажатие кнопки **Add** заносит пароль в список дополнительных паролей.

Нажатие кнопки **Change** переводит выбранный в списке пароль в режим редактирования, при этом появляются кнопки **Accept** (Подтвердить) и **Revert** (Возврат), а также разблокируется кнопка **Delete**. В процессе редактирования пароль можно изменить, удалить или оставить без изменений. После смены пароля и прав доступа внесенные изменения утверждаются нажатием кнопки **Accept**. Для выхода из режима редактирования и возврата к прежним установкам пароля необходимо нажать кнопку **Revert**. Удаляется пароль нажатием кнопки **Delete**.

Отметим, что из приложения паролями можно управлять с помощью методов компонента `Session`. Управление паролями заключается в их добавлении и удалении. Процедура `AddPassword(const Password: String)` добавляет новый пароль, заданный параметром `Password`; процедура `RemovePassword(const Password: String)` удаляет указанный пароль, а процедура `RemoveAllPasswords` удаляет все пароли.

Задание языкового драйвера

Для задания языкового драйвера нужно выбрать пункт **Table Language** (Язык таблицы) комбинированного списка **Table properties** окна определения структуры таблицы (см. рис. 16.2). При этом под списком становится доступной кнопка **Modify**, открывающая окно **Table Language** (рис. 6.12). Под "языком" таблицы понимается языковой драйвер, используемый для этой таблицы. В поле списка **Language** отображается текущий драйвер.

В списке можно выбрать драйвер нужного языка, для русского языка это драйвер Pdox ANSI Cyrillic, который корректно отображает символы русского алфавита и выполняет с ними операции сортировки.

По умолчанию языковой драйвер определяется установками процессора баз данных BDE, поэтому целесообразно установить его (параметр `LANGDRIVER`) в нужное значение,

например, с помощью программы Administrator BDE. Параметры BDE и вопросы, связанные с их настройкой, рассматриваются в *главе 22*.

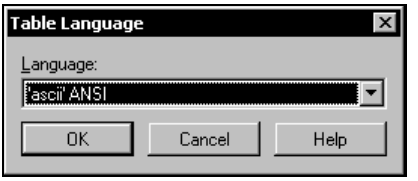


Рис. 16.12. Выбор языкового драйвера

Задание таблицы для выбора значений

Часто возникает ситуация, когда в поле должны заноситься значения из какого-либо набора, который может формироваться различными способами. Одним из часто используемых является вариант, когда эти значения содержатся в поле другой таблицы, а совокупность значений всех записей этого поля образует набор допустимых значений.

Для полей некоторых типов, например, строкового, числового или даты, можно определить некоторую таблицу *(таблицу выбора)*, поля которой будут использоваться для формирования набора допустимых значений. Если для поля задана таблица выбора, то в него можно ввести только значение, содержащееся в таблице выбора (в указанном поле любой записи). Задание таблицы выбора гарантирует, что в поле не будет введено недопустимое значение.

Действие набора допустимых значений распространяется также на редактирование записей таблицы программным способом: при попытке присвоить полю недопустимое значение генерируется исключение.

Замечание

При вводе строковых значений учитывается регистр букв, поэтому, например, `Водитель` и `ВОДИТЕЛЬ` являются разными значениями.

В любом случае допустимым является пустое значение (`Null`), если, конечно, не задано ограничение, что поле не может быть пустым.

Для выполнения операций, связанных с полями выбора, предназначен пункт **Table Lookup** (Таблица выбора) комбинированного списка **Table properties** окна определения структуры таблицы (см. рис. 16.2). При этом под списком становится доступной кнопка **Define**, нажатие которой открывает окно **Table Lookup** (рис. 16.13).

В списке **Fields** выводятся имена всех полей таблицы, при этом имена полей, которые не допускают создание таблицы выбора, выделены серым цветом (например, поле автоинкрементного типа).

Имя поля, для которого задается таблица выбора, отображается в области **Field name**, для его указания следует выделить в списке **Fields** нужную строку и нажать кнопку с изображением стрелки вправо. Если указать другое поле и нажать кнопку, то имя этого поля перейдет в область **Field name** и заменит имя предыдущего поля.

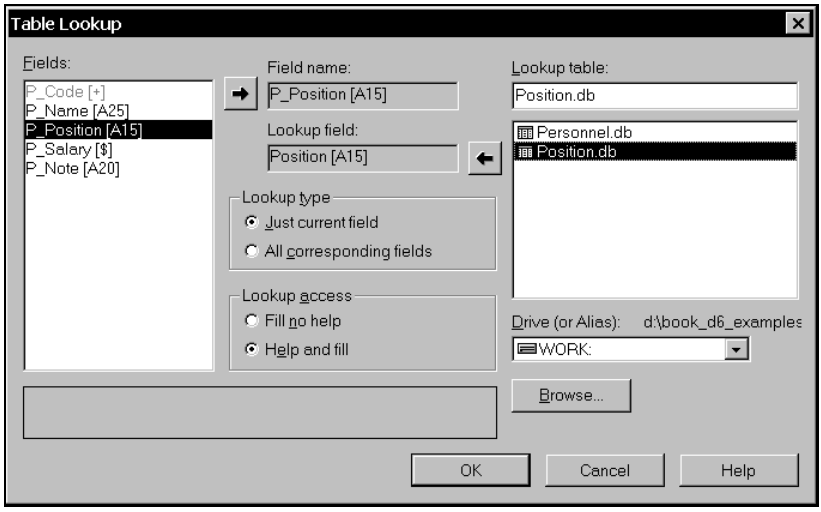


Рис. 16.13. Окно задания таблицы выбора

В списке **Lookup table** задается таблица выбора, имя которой (имя главного файла) появляется в поле редактирования над списком. В списке **Drive (or Alias)** задается диск или псевдоним, которые определяют местоположение таблицы выбора. Название каталога, таблицы которого содержатся в списке **Lookup table**, отображается справа от названия списка **Drive (or Alias)**. Таблицу также можно задать в окне **Select File**, открываемом нажатием кнопки **Browse** (Просмотр).

После задания таблицы выбора нажатие кнопки с изображением стрелки влево переводит имя *ее первого поля*, значения которого будут использованы для формирования набора допустимых значений, в область **Lookup field** (Поле выбора). Типы полей обеих таблиц должны совпадать, в противном случае в информационной панели выдается сообщение об ошибке.

С помощью группы переключателей **Lookup type** (Тип выбора) можно задать способ взаимодействия обеих таблиц. Если включен переключатель **Just current field** (Только текущее поле), то таблица выбора задается только для поля, указанного в области **Field name**. Переключатель **All corresponding fields** назначает таблицу выбора не только указанному полю, но и всем соответствующим полям. Имена и типы этих полей должны совпадать с соответствующими полями таблицы выбора.

Группа переключателей **Lookup access** (Доступ к таблице выбора) определяет, как пользователь использует значения из таблицы выбора. Если включен переключатель **Fill no help** (Вставка без помощи), то при редактировании поля, для которого определена таблица выбора, пользователь должен знать допустимые значения этого поля. При этом термин "выбор" не совсем точно отражает взаимосвязь полей таблиц, т. к. пользователю не предлагается список возможных значений, из которых он может выбрать нужное ему. Пользователь сам вводит значение в поле, при этом на уровне таблицы выполняется проверка, является ли это значение допустимым. Если значение отсутствует в указанном поле таблицы выбора, то оно не принимается.

В этом случае может помочь отображение рядом с редактируемой таблицей таблицы выбора, например, как показано на рис. 16.14.

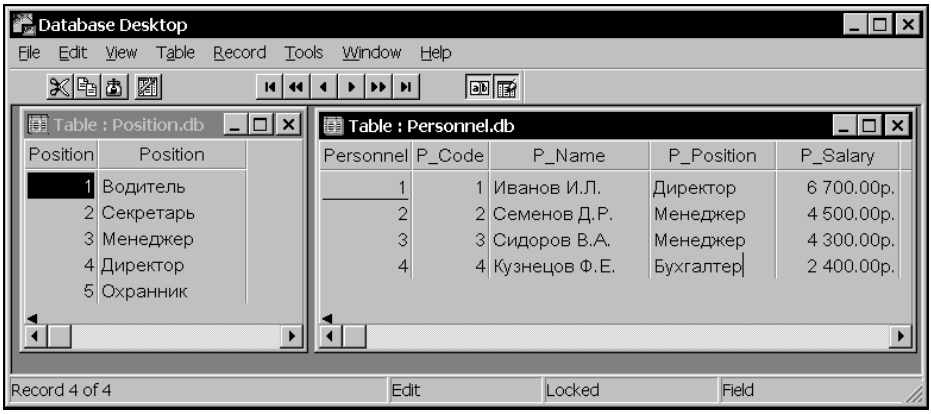


Рис. 16.14. Функционирование поля, для которого задана таблица выбора

В приведенном на этом рисунке примере в окне программы Database Desktop таблица `Personnel` содержит данные о сотрудниках организации, в ней определены поля для кода (`P_Code`), фамилии (`P_Name`), должности (`P_Position`), оклада (`P_Salary`) и примечания (`P_Note`). В строковое поле должности можно вводить только допустимое название должности. Перечень всех должностей содержится в единственном поле `Position` таблицы `Position`. Для предотвращения ввода в таблицу `Personnel` неправильных значений ее полю `P_Position` назначена таблица выбора `Position` с полем `Position`. В таблице `Personnel` для первых трех сотрудников введены должности, имеющиеся в таблице `Position`. Попытка задать для четвертого сотрудника должность `Бухгалтер` блокируется, т. к. эта должность не является разрешенной для ввода. При необходимости эту должность сначала нужно ввести в таблицу `Position`, после чего она станет доступной для таблицы `Personnel`.

Установка переключателя **Help and fill** (Помощь и вставка) позволяет не только ввести значение в поле, как описано выше, но и действительно выбрать значение в списке. Список, сформированный на основании значений поля (полей) таблицы выбора (рис. 16.15), появляется при нажатии комбинации клавиш <Ctrl>+<Пробел> в редактируемом поле. После выбора нужного значения и нажатия кнопки **OK** оно заносится в поле.

Замечание

Из таблицы выбора могут быть удалены значения. Если эти значения содержатся в полях таблицы, использующей таблицу выбора, то при переходе в режим ее редактирования возникает ошибка. При использовании программы типа Database Desktop таблицу нельзя вывести из режима редактирования, пока все ее значения не приведены в соответствие с новыми значениями таблицы выбора. Во время выполнения приложения при доступе к таблице, использующей удаленные из таблицы выбора значения, генерируется исключение.

После задания для поля таблицы выбора в окне определения структуры таблицы (см. рис. 16.2) становятся доступными кнопки **Modify** и **Erase**, а также список, в котором отображается имя этой таблицы.

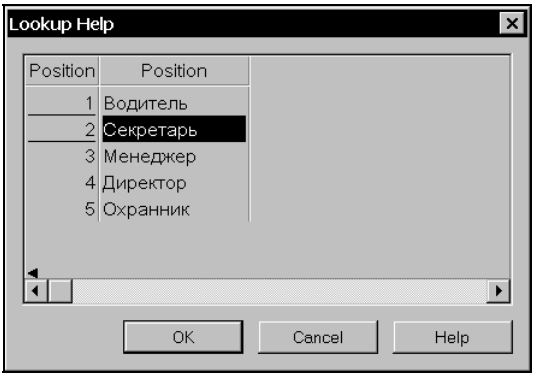


Рис. 16.15. Список выбора

Нажатие кнопки **Modify** вновь открывает окно задания таблицы выбора, в котором можно ввести новые данные. Кнопка **Erase** служит для отмены использования таблицы выбора и связанных с ней ограничений на значения поля.

Отметим, что объекты поля типа `TField` наборов данных `Table` и `Query`, а также компонент `DBGrid` позволяют определить для поля список выбора, который также дает возможность пользователю выбирать значения при редактировании. Они задаются на программном уровне и действуют только в своем приложении. Определение и использование подобных списков выбора рассматриваются в *следующих двух главах*.

Просмотр списка подчиненных таблиц

Таблица, связанная с другими таблицами, является либо главной, либо подчиненной. Выше мы рассмотрели задание ссылочной целостности для подчиненной таблицы. Для главной таблицы можно просмотреть список подчиненных таблиц, отображаемый при выборе пункта **Dependent Table** (Подчиненная таблица) все того же комбинированного списка **Table properties** (см. рис. 16.2). Этот список содержит имена всех таблиц, имеющих условия ссылочной целостности с участием данной таблицы.

Изменение структуры таблицы

Структуру существующей таблицы можно изменить, выполнив команду **Table | Restructure** после предварительного выбора таблицы в окне программы Database Desktop. В результате открывается окно определения структуры таблицы, и дальнейшие действия не отличаются от действий, выполняемых при создании таблицы.

Замечание

При изменении структуры таблицы с ней не должны работать другие приложения, в том числе Delphi. Поэтому предварительно необходимо закрыть Delphi или приложение, в котором компоненты `Table` связаны с перестраиваемой таблицей. Другим вариантом является отключение активности компонентов `Table`, связанных с перестраиваемой таблицей, для чего свойству `Active` этих компонентов через Инспектор объектов устанавливается значение `False`.

Переименование таблицы следует выполнять из среды программы Database Desktop, а не из среды Windows, например, с помощью Проводника. Для этого при работе со структурой

таблицы можно нажать кнопку **Save as** и задать новое имя таблицы. В результате в указанном каталоге диска появятся все необходимые файлы таблицы. При этом старая таблица также сохраняется. Информация о названии таблицы используется внутри ее файлов, поэтому простое переименование всех файлов таблицы приведет к ошибке при попытке доступа к ней.

Если необходимо просто ознакомиться со структурой таблицы, то выполняется команда **Table | Into Structure1**. В результате появляется окно определения структуры таблицы, но элементы, с помощью которых в структуру таблицы могут быть внесены изменения, заблокированы. Просмотр структуры возможен также для таблицы, с которой связаны другие приложения.

Характеристика приложения для работы с базами данных

При создании приложения для работы с базами данных с применением любой технологии доступа к данным используется стандартный состав и схема связи компонентов и таблицы базы данных. В состав приложения Delphi для работы с базами данных входят три типа компонентов:

- источник данных;
- визуальные компоненты;
- наборы данных.

Компонент-источник данных играет роль связующего звена между набором данных и визуальными компонентами. *Визуальные компоненты* служат для навигации по набору данных, отображения и редактирования записей.

Наборы данных служат для организации связи с таблицами базы данных. В Delphi для разных механизмов доступа к данным в качестве наборов данных используются свои компоненты.

Взаимосвязь компонентов приложения с таблицей, БД и используемые при этом свойства компонентов показаны на рис. 16.16.

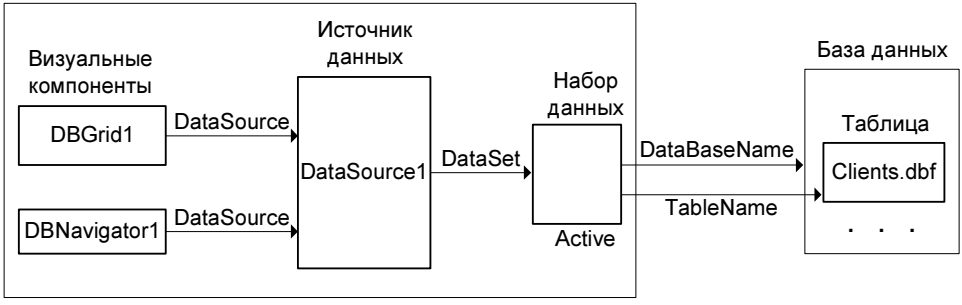


Рис. 16.16. Взаимосвязь компонентов приложения, таблицы и БД

Разрабатывая приложение, можно задавать значения всех свойств компонентов с помощью Инспектора объектов. При этом требуемые значения либо непосредственно

вводятся в поле, либо выбираются в раскрывающихся списках. В последнем случае приложение создается с помощью мыши и не требует набора каких-либо символов на клавиатуре.

Для примера в табл. 16.1 приведены компоненты, используемые для доступа к данным таблицы Clients.dbf базы данных `dbdemos` с помощью механизма BDE, их основные свойства и значения этих свойств. В качестве набора данных здесь используется компонент `Table1`.

В дальнейшем при организации приложений, использующих механизм доступа BDE, предполагается, что названные компоненты связаны между собой именно таким образом, и свойства, с помощью которых эта связь осуществляется, не рассматриваются. Для приложений, использующих другие механизмы доступа, связь между компонентами устанавливается аналогично.

Таблица 16.1. *Значения свойств компонентов*

Компонент	Свойства	Значения

`Table1`	`DataBaseName`
	`TableName`
	`Active`
		`dbdemos`
		`Clients.dbf`
		`True`

`DataSource1`	`DataSet`	`Table1`
`DBGrid1`	`DataSource`	`DataSource1`
`DBNavigator1`	`DataSource`	`DataSource1`

Для примера рассмотрим вид формы приложения, использующего механизм доступа BDE и позволяющего перемещаться по записям таблицы БД, просматривать и редактировать поля, удалять записи из таблицы, а также вставлять новые. Файл проекта приложения обычно не требует от разработчика выполнения каких-либо действий. Поэтому при создании приложения главной задачей является конструирование форм, в простейшем случае — одной формы.

Вид формы приложения на этапе проектирования показан на рис. 16.17, где в форме размещены компоненты `Table1`, `DataSource1`, `DBGrid1` и `DBNavigator1`.

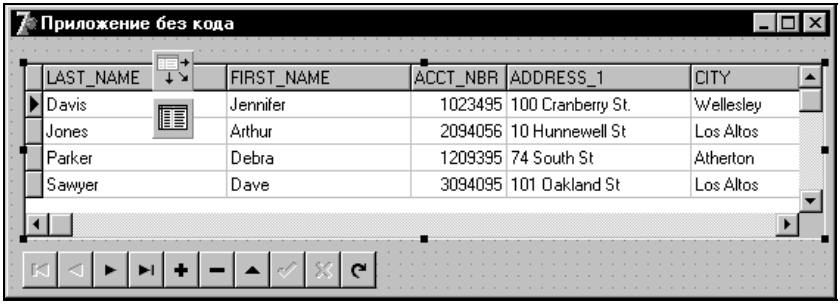


Рис. 16.17. Форма приложения для работы с БД

Компонент `Table1` обеспечивает взаимодействие с таблицей БД. Для связи с требуемой таблицей нужно установить в соответствующие значения свойство `DataBaseName`, ука-

зывающее путь к БД, и свойство `TableName`, указывающее имя таблицы. После задания таблицы для открытия набора данных свойство `Active` должно быть установлено в значение `True`.

Замечание

Значение `True` свойства `Active` нужно устанавливать после задания таблицы БД, т. е. после установки нужных значений свойств `DataBaseName` и `TableName`.

Имя таблицы лучше выбирать в раскрывающемся списке в поле значения свойства `TableName`. Если путь к БД (свойство `DataBaseName`) задан правильно, то в этом списке отображаются главные файлы всех доступных таблиц.

В рассматриваемом приложении использована таблица клиентов, входящая в состав поставляемых с Delphi примеров, ее главный файл — Clients.dbf. Файлы этой и других таблиц примеров находятся в каталоге, путь к которому указывает псевдоним `dbdemos`. Настройка псевдонима может быть выполнена с помощью программы BDE Administrator.

Компонент `DataSource1` является промежуточным звеном между компонентом `Table1`, соединенным с реальной таблицей БД, и визуальными компонентами `DBGrid1` и `DBNavigator1`, с помощью которых пользователь взаимодействует с этой таблицей. На компонент `Table1`, с которым связан компонент `DataSource1`, указывает свойство `DataSet` последнего.

Компонент `DBGrid1` отображает содержимое таблицы БД в виде сетки, в которой столбцы соответствуют полям, а строки — записям таблицы. По умолчанию пользователь может просматривать и редактировать данные. Компонент `DBNavigator1` позволяет пользователю перемещаться по таблице, редактировать, вставлять и удалять записи. Компоненты `DBGrid1` и `DBNavigator1` связываются со своим источником данных — компонентом `DataSource1` — через свойства `DataSource`.

Взаимосвязь компонентов приложения и таблицы БД и используемые при этом свойства компонентов показаны на рис. 16.16. Компоненты, используемые для работы с таблицей БД, их основные свойства и значения этих свойств приведены в табл. 16.1.

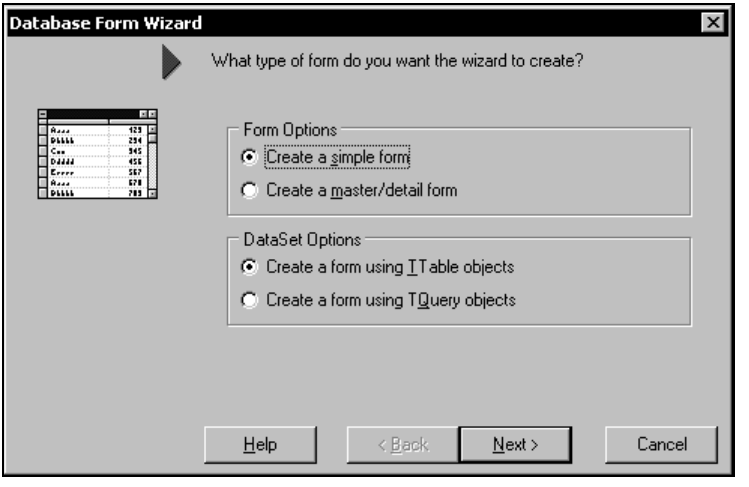


Рис. 16.18. Окно Мастера Database Form Wizard

Для автоматизации процесса создания формы, использующей компоненты для операций с БД, можно вызвать **Database Form Wizard** (Мастер форм баз данных), показанный на рис. 16.18. Этот Мастер расположен на странице **Business** Хранилища объектов.

Мастер позволяет создавать формы для работы с отдельной таблицей и со связанными таблицами, при этом можно использовать наборы данных `Table` или `Query`.

Использование модуля данных

При конструировании формы невизуальные компоненты, используемые для доступа к данным, такие как `DataSource` или `Table`, размещаются в форме, но при выполнении приложения эти компоненты не видны. Поэтому их можно размещать в любом удобном месте формы, которая для них является контейнером — модулем. Кроме того, для размещения невизуальных компонентов, через которые осуществляется доступ к данным, предназначен специальный объект — *модуль данных*.

Есть три типа модулей данных:

- простой модуль данных;
- удаленный модуль данных;
- Web-модуль.

Далее рассматривается *простой модуль* данных, который представлен объектом `DataModule`. Использование *удаленного модуля* данных и *Web-модуля* подробно изучается при рассмотрении трехуровневых приложений и публикации БД в Интернете *(см. главы 29, 30, 33)*.

Если применяется простой модуль данных, то взаимосвязь компонентов приложения и таблицы БД имеет вид, показанный на рис. 16.19.

Модуль данных, как и форма, является контейнером для своих невизуальных компонентов, и для него создается модуль кода с расширением pas. Добавление модуля данных к проекту выполняется командой **File | New | DataModule** главного меню Delphi. В окне модуля компоненты размещаются так же, как и в форме (рис. 16.20). При выборе объекта в окне Инспектора объектов отображаются его свойства, значения которых можно просматривать и изменять.

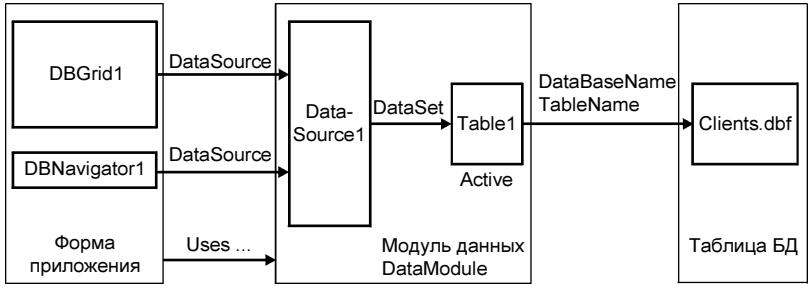


Рис. 16.19. Взаимосвязь компонентов приложения и таблицы БД при использовании модуля данных

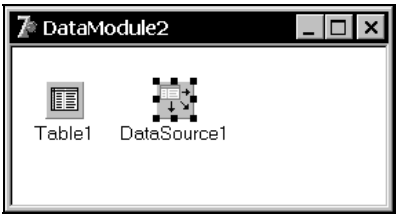


Рис. 16.20. Модуль данных

При обращении к содержащимся в модуле данных компонентам для них указывается составное имя, в которое, кроме имени компонента, входит имя модуля данных. Составное имя имеет формат

```
<Имя модуля данных>.<Имя компонента>
```

Далее приводится пример кода, в котором осуществляется обращение к компонентам модуля данных.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DataModule2.Table1.DatabaseName := 'dbdemos';
  DataModule2.Table1.TableName := 'Clients.dbf';
  DataModule2.DataSource1.DataSet := DataModule2.Table1;
  DBGrid1.DataSource := DataModule2.DataSource1;
  DBNavigator1.DataSource := DataModule2.DataSource1;
  DataModule2.Table1.Active := True;
end;
```

Для компонентов устанавливаются значения свойств, связывающих между собой эти компоненты и таблицу БД. Значения свойств устанавливаются *динамически* в процессе выполнения приложения, для чего использован обработчик события создания главной формы приложения. В составных именах компонентов доступа к данным, которыми являются источник данных `DataSource1` и набор данных `Table1`, указывается имя модуля данных `DataModule2`.

Чтобы обеспечить возможность доступа к компонентам модуля данных в модуле формы, в список `uses` раздела `implementation` необходимо включить ссылку на модуль данных:

```
uses unit2;
```

Ссылку на другой модуль можно написать самостоятельно, но Delphi позволяет вставить ее автоматически. При выборе команды **File | Use Unit** появляется диалоговое окно **Use Unit**. После выбора нужного модуля и нажатия кнопки **OK** он добавляется в список.

Если ссылка на требуемый модуль отсутствует, но в коде используется имя модуля данных, то при компиляции приложения появляется диалоговое окно **Information** (рис. 16.21). Оно сообщает, что форма ссылается на другую форму, объявленную в модуле, отсутствующем в списке `uses` формы. Для автоматического добавления модуля в список достаточно нажать кнопку **Yes**.

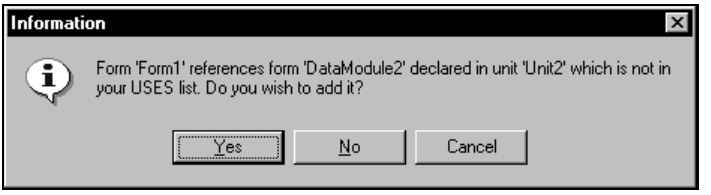


Рис. 16.21. Диалоговое окно **Information** позволяет добавить отсутствующий модуль в список `uses`

Помимо компонентов доступа к данным, которыми являются `Session`, `Database`, `Table`, `Query`, `StoredProc`, `BatchMove` и др., в модуле данных можно размещать невизуальные компоненты, не имеющие прямого отношения к БД, например, `ImageList`, `OpenDialog` или `Timer`.

Замечание

При работе с модулем данных в Палитре компонентов доступны только невизуальные компоненты.

Модуль данных позволяет:

- отделить управление БД от обработки данных;
- создать модуль, совместно используемый несколькими приложениями.

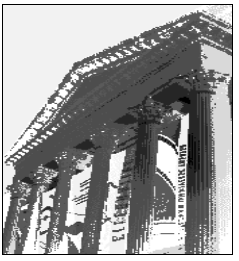
Основным назначением модуля данных является *централизованное хранение* компонентов доступа к данным, а также кода для этих компонентов, в частности, обработчиков событий. В модуле данных удобно размещать код, выполняющий управление БД, например, реализацию бизнес-правил.

Использование простого модуля данных несколькими приложениями позволяет ускорить разработку приложений, т. к. готовый модуль данных впоследствии можно включать в новые приложения. Кроме того, управление БД через общий модуль дает возможность определить для всех пользователей одинаковые режимы и правила работы с базой, а также делает более простым изменение этих режимов и правил.

Однако для небольших приложений использование простого модуля данных не всегда оправданно, т. к. может затруднить, а не облегчить разработку приложения.

Удаленный модуль данных предназначен для работы с удаленными БД в трехуровневой архитектуре "клиент-сервер" и используется для создания *сервера приложения* — промежуточного уровня между приложением и сервером БД.

Web-модуль предназначен для работы с БД в сети Интернет и является посредником между обозревателем (программой просмотра Web-документов) и сервером БД. Использование Web-модуля рассмотрено в *главе 33*, посвященной публикации баз данных средствами Delphi.



ГЛАВА 17

Компоненты доступа к данным

Компоненты доступа к данным являются невизуальными. В этой главе мы рассмотрим основные компоненты доступа к данным, которые используются при работе с локальными и удаленными БД с помощью механизма BDE. Компоненты `Session` и `Database`, применяемые для управления соединениями с БД и транзакциями, будут изучены в *главах 19* и *26*.

Наборы данных

Таблицы БД располагаются на диске и являются физическими объектами. Для операций с данными, содержащимися в таблицах, используются наборы данных. В терминах системы Delphi *набор данных* представляет собой совокупность записей, взятых из одной или нескольких таблиц БД. Записи, входящие в набор данных, отбираются по определенным правилам, при этом в частных случаях набор данных может включать в себя все записи из связанной с ним таблицы или не содержать ни одной записи. Набор данных является *логической таблицей*, с которой можно работать при выполнении приложения. Взаимодействие таблицы и набора данных напоминает взаимодействие физического файла и файловой переменной.

Замечание

В отличие от Delphi, многие СУБД вместо термина *"набор данных"* используют термины *"выборка"* или *"таблица"*.

В Delphi для работы с наборами данных при использовании механизма BDE служат такие компоненты, как `Table`, `Query`, `UpdateSQL`, `DecisionQuery` или `StoredProc`. В случае других механизмов доступа для работы с наборами данных служат аналогичные компоненты `ADOTable`, `ADOQuery` и `ADOStoredProc` (для механизма ADO), `SQLTable`, `SQLQuery` и `SQLStoredProc` (для механизма dbExpress), `IBTable`, `IBQuery`, `IBUpdateSQL` и `IBStoredProc` (для механизма InterBease). Отличительные особенности последних мы рассмотрим при описании соответствующих механизмов доступа. Здесь остановимся на изучении компонентов для механизма доступа BDE.

Компонент `StoredProc` используется для вызова хранимых процедур при организации взаимодействия с удаленными БД, а компонент `UpdateSQL` обеспечивает работу с кэши-

рованными изменениями в записях. Эти компоненты рассматриваются при описании удаленных БД. Компонент DecisionQuery применяется при построении систем принятия решений. Наиболее универсальными и, соответственно, часто используемыми являются компоненты Table и Query, задающие наборы данных. Они будут подробно описаны немного позже.

Базовые возможности доступа к БД обеспечивает класс TDataSet, представляющий наборы данных в виде совокупности строк и столбцов (записей и полей). Этот класс содержит основные средства навигации (перемещения) и редактирования наборов данных.

Компоненты Table и Query являются производными от класса TDBDataSet — потомка класса TDataSet (через класс TBDEDataSet). Они демонстрируют схожие с базовыми классами характеристики и поведение, но каждый из них имеет и свои особенности. Здесь мы рассмотрим наиболее общие характеристики наборов данных. Большая часть свойств, методов и событий изучается на примере операций с наборами данных.

Расположение БД, с таблицами которой выполняются операции, указывает свойство DatabaseName типа String. Значением свойства является имя каталога, в котором находится БД (файлы ее таблиц), или псевдоним, ссылающийся на этот каталог. Если для БД определен псевдоним, то его можно выбрать в раскрывающемся списке окна Инспектора объектов.

Замечание

Желательно задавать имя БД через псевдоним. Это заметно облегчает перенос приложения и файлов БД в другие каталоги и на другие компьютеры, т. к. для обеспечения работоспособности приложения после изменения расположения БД достаточно изменить название каталога, на который ссылается псевдоним БД.

Для компонента Table использование свойства DatabaseName является единственной возможностью задать местонахождение таблиц БД. Для компонента Query дополнительно можно указать в запросе SQL путь доступа к каждой таблице.

Замечание

При задании расположения БД программным способом набор данных предварительно необходимо закрыть, установив его свойство Active в значение False. В противном случае генерируется исключение.

Вот пример, иллюстрирующий, как задается расположение БД:

```
Table1.Active := False;
Table1.DatabaseName := 'BDPlace';
Table2.Active := False;
Table2.DatabaseName := 'C:\SALE\BD';
```

Для набора данных Table1 таблицы БД расположены в каталоге, на который указывает псевдоним BDPlace. Таблицы БД для набора данных Table2 расположены в каталоге C:\SALE\BD. Для определения и изменения псевдонима и его параметров удобно использовать такие программы, как Database Desktop или BDE Administrator (рассматриваются в *главе 24*).

В зависимости от ограничений и критерия фильтрации один и тот же набор данных в разные моменты времени может содержать различные записи. Число записей, составляющих набор данных, определяет свойство `RecordCount` типа `Longint`. Это свойство доступно для чтения при выполнении приложения. Управление числом записей в наборе данных осуществляется косвенно — путем отбора записей тем или иным способом, например, с помощью фильтрации или SQL-запроса (для компонента `Query`).

В приводимом примере производится перебор всех записей набора данных:

```
var i: integer;
...
Table1.First;
for i := 1 to Table1.RecordCount do begin
    // Здесь можно расположить инструкции, выполняющие
    // обработку очередной записи
    Table1.Next;
end;
```

Перебор всех записей набора данных осуществляется в цикле, для чего переменная `i` цикла последовательно принимает значения от `1` до `RecordCount`. Перед началом цикла вызовом метода `First` выполняется переход к первой записи набора данных. В цикле для перехода к следующей записи вызывается метод `Next`.

Для локальных таблиц dBase или Paradox составляющие набор данных записи последовательно нумеруются, отсчет начинается с единицы. Номер записи в наборе данных определяет свойство `RecNo` типа `Longint`, которое доступно во время выполнения программы.

Номер текущей записи можно узнать, например, так:

```
Edit1.Text := IntToStr(Table1.RecNo);
```

Замечание

При изменении порядка записей при сортировке или фильтрации нумерация записей также изменяется.

Для таблиц Paradox свойство `RecNo` можно использовать для перехода к требуемой записи, установив в качестве значения свойства номер записи. Так, в операции

```
Table1.RecNo := StrToInt(Edit1.Text);
```

выполняется переход к записи, номер которой содержится в поле ввода `Edit1`. Указанная запись становится текущей.

Для выполнения операций с наборами данных используются два способа доступа к данным:

- навигационный;
- реляционный.

Навигационный способ доступа заключается в обработке каждой *отдельной записи* набора данных. Этот способ обычно используется в локальных БД или в удаленных БД небольшого размера. При навигационном способе доступа каждый набор данных имеет невидимый указатель текущей записи. Указатель определяет запись, с которой могут выполняться такие операции, как редактирование или удаление. Поля текущей записи

доступны для просмотра. Например, компоненты DBEdit и DBText отображают содержимое соответствующих полей именно текущей записи. Компонент DBGrid указывает текущую запись с помощью специального маркера.

В разрабатываемом приложении навигационный способ доступа к данным можно реализовать, используя любой из компонентов Table или Query.

Реляционный способ доступа основан на обработке *группы записей*. Если требуется обработать одну запись, все равно обрабатывается группа, состоящая из одной записи. При реляционном способе доступа используются SQL-запросы, поэтому его называют также *SQL-ориентированным*. Реляционный способ доступа ориентирован на работу с удаленными БД и является для них предпочтительным. Однако его можно использовать и для локальных БД.

Реляционный способ доступа к данным в приложении можно реализовать с помощью компонента Query.

Состояния наборов данных

Наборы данных могут находиться в *открытом* или *закрытом* состояниях, на что указывает свойство Active типа Boolean. Если свойству Active установлено значение True, то набор данных открыт. Открытый компонент Table содержит набор данных, соответствующий данным таблицы, связанной с ним через свойство TableName. Для открытого компонента Query набор данных соответствует результатам выполнения SQL-запроса, содержащегося в свойстве SQL этого компонента. Если свойство Active имеет значение False (по умолчанию), то набор данных закрыт, и его связь с БД разорвана.

Набор данных может быть открыт на этапе разработки приложения. Если при этом к набору данных через источник данных DataSource подключены визуальные компоненты, например, DBGrid или DBEdit, то они отображают соответствующие данные таблицы БД.

Замечание

На этапе проектирования приложения визуальные компоненты отображают данные записей набора данных, но перемещение по набору данных и редактирование записей невозможны. Исключение составляет возможность перемещения текущего указателя с помощью полос прокрутки компонента DBGrid.

Если по каким-либо причинам открыть набор данных невозможно, то при попытке установить свойство Active в значение True выдается сообщение об ошибке, а свойство Active сохраняет значение False. Одной из причин невозможности открытия набора данных может являться неправильное значение свойства TableName или SQL.

Замечание

На этапе проектирования свойство Active наборов данных автоматически устанавливается в значение False при изменении значения свойств DataBaseName, TableName или SQL.

Приводимый пример демонстрирует управление состоянием набора данных с помощью свойства Active, которое используется для открытия и закрытия набора данных Query1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Query1.Active := False;
  Query1.SQL.Clear;
  Query1.SQL.Add('select * from Example1.db');
  Query1.Active := True;
end;
```

Управлять состоянием набора данных можно также с помощью методов `Open` и `Close`.

Процедура `Open` открывает набор данных, ее вызов эквивалентен установке свойства `Active` в значение `True`. При вызове метода `Open` генерируются события `BeforeOpen` и `AfterOpen`, а также вызываются процедуры-обработчики этих событий.

Процедура `Close` закрывает набор данных, ее вызов эквивалентен установке свойства `Active` в значение `False`. При вызове метода `Close` генерируются события `BeforeClose` и `AfterClose`, а также вызываются процедуры-обработчики этих событий.

В примере показывается управление состоянием набора данных (открытие и закрытие) с помощью методов `Open` и `Close`:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Query1.Close;
  Query1.SQL.Clear;
  Query1.SQL.Add('select * from Example2.db');
  Query1.Open;
end;
```

События `BeforeOpen` и `AfterOpen` имеют тип `TDataSetNotifyEvent`, который описывается так:

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
```

В этом описании параметр `DataSet` определяет набор данных, для которого произошло событие.

Событие `BeforeOpen` возникает непосредственно перед открытием набора данных. В обработчике этого события можно выполнить проверку определенных условий, и если они не соблюдаются, то открытие набора данных может быть запрещено.

Замечание

Обработчик события не содержит специального параметра, с помощью которого можно запретить открытие набора данных. Одним из вариантов запрета открытия является принудительное возбуждение исключения.

Рассмотрим в качестве примера следующую процедуру:

```
procedure TForm1.Table1BeforeOpen(DataSet: TDataSet);
begin
  if not CheckBox1.Checked then Abort;
end;
```

Если флажок `CheckBox1`, управляющий возможностью открытия набора данных, не установлен, то открытие набора данных `Table1` блокируется. Для этого с помощью вы-

зова процедуры Abort генерируется "тихое" исключение. В результате операции, связанные с открытием набора данных, отменяются, а пользователю не выдается никаких сообщений об ошибках. В подобных случаях выдачу предупреждающих сообщений должен обеспечивать программист, как это реализуется, например, в приведенной далее процедуре:

```
procedure TForm1.Table1BeforeOpen(DataSet: TDataSet);
begin
  if not CheckBox1.Checked then begin
    MessageDlg('Данные таблицы ' + Table1.TableName + ' недоступны!',
                mtError, [mbOK], 0);
    Abort;
  end;
end;
```

Событие AfterOpen генерируется сразу после открытия набора данных. Это событие можно использовать, например, для выдачи пользователю сообщения о возможности работы с данными.

В примере демонстрируется открытие набора данных с выдачей соответствующего сообщения:

```
procedure TForm1.Table1AfterOpen(DataSet: TDataSet);
begin
  if CheckBox2.Checked then
     MessageDlg('Данные таблицы ' + Table1.TableName +
                ' доступны для работы.',
                  mtWarning, [mbOK], 0);
end;
```

Если флажок CheckBox2, управляющий возможностью выдачи сообщений, установлен, то при открытии набора данных Table1 пользователю выдается сообщение.

Как уже говорилось, при закрытии набора данных возникают события BeforeClose и AfterClose. Они, как и события BeforeOpen и AfterOpen, имеют тип TDataSetNotifyEvent.

Отметим, что закрытие набора данных автоматически не сохраняет текущую запись, т. е. если набор данных при закрытии находился в режимах редактирования или вставки, то произведенные изменения данных в текущей записи будут потеряны. Поэтому перед закрытием набора данных нужно проверять его режим и при необходимости принудительно вызывать метод Post, сохраняющий сделанные изменения. Одним из вариантов сохранения изменений является вызов метода Post в обработчике события BeforeClose, возникающего непосредственно перед закрытием набора данных.

Рассмотрим следующий пример:

```
procedure TForm1.Table1BeforeClose(DataSet: TDataSet);
begin
  if (Table1.State = dsEdit) or (Table1.State = dsInsert) then Table1.Post;
end;
```

Если набор данных Table1 находится в режиме редактирования или вставки, то перед его закрытием внесенные изменения сохраняются.

Замечание

При закрытии приложения событие `BeforeClose` не генерируется и несохраненные изменения теряются.

Событие `AfterClose` возникает сразу после закрытия набора данных, и его можно использовать для выдачи пользователю соответствующих сообщений, как это сделано в приведенной далее процедуре:

```
procedure TForm1.Table1AfterClose(DataSet: TDataSet);
begin
  if CheckBox2.Checked then begin
    Beep;
    MessageDlg('Таблица ' + Table1.TableName + ' закрыта.',
               mtWarning, [mbOK], 0);
  end;
end;
```

Если установлен флажок `CheckBox2`, управляющий режимом выдачи сообщений, то после закрытия набора данных `Table1` выдается сообщение о закрытии таблицы, связанной с этим набором данных.

Необходимо иметь в виду, что если при работе приложения используется большое число таблиц, то выдача подобных сообщений может затруднять действия пользователя. Поэтому программист должен предусмотреть возможность отключения выдачи сообщений. В приведенных примерах для этой цели предназначен флажок `CheckBox2`.

Режимы наборов данных

Наборы данных могут находиться в различных режимах. Текущий режим набора данных определяется свойством `State` типа `TDataSetState`. Оно доступно для чтения во время выполнения приложения и может быть использовано только для текущего режима. Для перевода набора данных в требуемый режим используются специальные методы. Они могут вызываться явно (указанием имени метода) или косвенно (путем управления соответствующими визуальными компонентами, например, навигатором `DBNavigator` или сеткой `DBGrid`).

Набор данных может находиться в одном из перечисленных далее режимов.

- `dsInactive` (неактивен) — набор данных закрыт и доступ к его данным невозможен. В этот режим набор данных переходит после своего закрытия, когда свойство `Active` установлено в значение `False`.
- `dsBrowse` (навигация по записям набора данных и просмотр данных) — в этот режим набор данных переходит так:
  - из режима `dsInactive` — при установке свойства `Active` в значение `True`;
  - из режима `dsEdit` — при вызове метода `Post` или `Cancel`;
  - из режима `dsInsert` — при вызове метода `Post` или `Cancel`.
- `dsEdit` (редактирование текущей записи) — в этот режим набор данных переходит из режима `dsBrowse` при вызове метода `Edit`.

- `dsInsert` (вставка новой записи) — в этот режим набор данных переходит из режима `dsBrowse` при вызове методов `Insert`, `InsertRecord`, `Append` или `AppendRecord`.
- `dsSetKey` (поиск записи, удовлетворяющей заданному критерию) — в этот режим набор данных переходит из режима `dsBrowse` при вызове методов `SetKey`, `SetRangeXXX`, `FindKey`, `GotoKey`, `FindNearest` или `GotoNearest`. Возможен только для компонента `TTable`, т. к. для компонента `Query` отбор записей осуществляется средствами языка SQL.
- `dsCalcFields` (расчет вычисляемых полей) — используется обработчик события `OnCalcFields`.
- `dsFilter` (фильтрация записей) — в этот режим набор данных автоматически переходит из режима `dsBrowse` каждый раз, когда выполняется обработчик события `OnFilterRecord`. В режиме блокируются все попытки изменения записей. После завершения работы обработчика события `OnFilterRecord` набор данных автоматически переводится в режим `dsBrowse`.
- `dsNewValue` (обращение к значению свойства `TField.NewValue`).
- `dsOldValue` (обращение к значению свойства `TField.OldValue`).
- `dsCurValue` (обращение к значению свойства `TField.CurValue`).
- `dsBlockRead` (запрет изменения элементов управления и генерации событий при вызове метода `Next`).
- `dsInternalCalc` (указание на необходимость вычислять значения полей, свойство `TField.FieldKind` которых имеет значение `fkInternalCalc`).
- `dsOpening` (открытие набора данных).

Взаимосвязи между основными режимами наборов данных показаны на рис. 17.1, где приведены также некоторые методы и свойства, с помощью которых набор данных переходит из одного режима в другой.

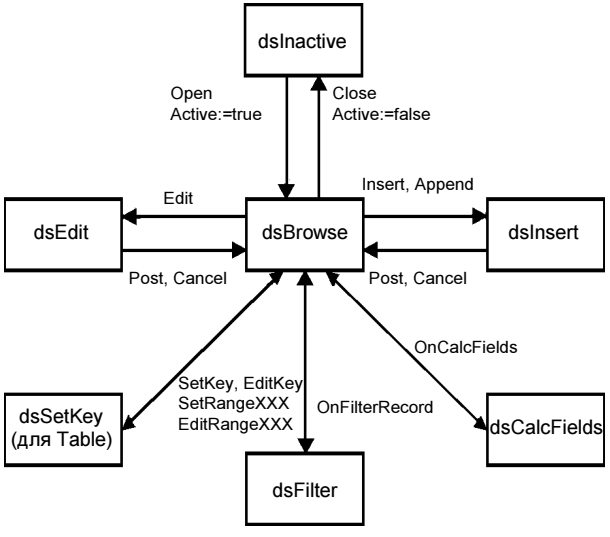


Рис. 17.1. Взаимосвязи режимов наборов данных

Иногда при описании операций, выполняемых с записями набора данных, под режимом *редактирования* подразумевается не только режим dsEdit изменения полей текущей записи, но и режим dsInsert вставки новой записи. Тем самым режим редактирования понимается в широком смысле слова как режим *модификации* набора данных.

Набор данных использует режимы dsNewValue, dsOldValue, dsCurValue, dsBlockRead и dsInternalCalc для собственных нужд, обычно программист не анализирует их.

При выполнении программы можно определить режим набора данных с помощью одноименных свойств State типа TDataSetState самого набора данных и связанного с ним источника данных DataSource. При изменении режима набора данных для источника данных DataSource генерируется событие OnStateChange типа TNotifyEvent.

Рассмотрим пример:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
begin
  case DataSource1.State of
    dsInactive: Label1.Caption := 'Набор данных закрыт';
    dsBrowse:   Label1.Caption := 'Просмотр набора данных';
    dsEdit:     Label1.Caption := 'Редактирование набора данных';
    dsInsert:   Label1.Caption := 'Вставка записи в набор данных'
    else        Label1.Caption := 'Режим набора данных не определен';
  end;
end;
```

В приведенной процедуре определяется режим набора данных, связанного с источником данных DataSource1, и информация об этом режиме выводится в надписи Label1. При этом используется свойство State источника данных. Код, выполняющий анализ режима, помещен в обработчик события OnStateChange компонента DataSource1.

Доступ к полям

Каждое поле набора данных представляет собой отдельный столбец, для работы с которым в Delphi служат объект Field типа TField и объекты производных от него типов, например, TIntegerField, TFloatField или TStringField. Для доступа к этим объектам и, соответственно, к полям записей у набора данных есть специальные методы и свойства, доступные при выполнении приложения.

Свойство FieldCount типа Integer указывает количество полей набора данных. Это свойство доступно только для чтения. Количество полей набора данных может отличаться от физического числа полей таблицы БД, поскольку в набор данных не обязательно включаются все поля таблицы. Состав полей формируется при разработке приложения с помощью Редактора полей набора данных и Редактора столбцов сетки DBGrid. Кроме того, возможно динамическое изменение состава полей во время выполнения приложения. Для компонента Query состав полей набора данных зависит также от SQL-запроса.

Значение свойства Fields[Index: Integer] типа TField представляет собой поле (столбец) набора данных. К отдельному полю можно обратиться, указав его номер Index в массиве Fields; номера полей находятся в пределах от нуля до FieldCount – 1. Номер объекта поля в массиве полей определяет свойство Index типа Integer. В отличие от

Редакторов полей и столбцов, применяемых на этапе разработки приложения, свойство `Index` можно использовать для определения и изменения порядка полей набора данных во время выполнения приложения.

В качестве примера рассмотрим чтение полей текущей записи:

```
procedure TForm1.Button1Click(Sender: TObject);
var n: integer;
begin
  for n := 0 to Table1.FieldCount – 1 do
    ListBox1.Items.Add(Table1.Fields[n].AsString);
end;
```

Здесь содержимое каждого поля текущей записи интерпретируется как строковое значение и добавляется к списку `ListBox1`.

Номер поля в наборе данных не является заранее фиксированным и известным числом и зависит от порядка полей в таблице БД, от текущего состава полей набора данных, а также от значений свойств некоторых визуальных компонентов, связанных с этим набором, например, сетки `DBGrid`. Так, при изменении порядка расположения столбцов в компоненте `DBGrid` соответственно изменяется порядок следования полей набора данных. В связи с этим обращение к какому-либо полю по его номеру в массиве полей может вызвать обращение совсем к другому полю. Поэтому для доступа к полям чаще используются методы `FindField` и `FieldByName`.

Функция `FindField(const FieldName: String): TField` возвращает для набора данных поле, имя которого указывает параметр `FieldName`. В отличие от номера в массиве полей, имя поля более статично и изменяется реже. Кроме того, имя поля несет бóльшую смысловую нагрузку, чем номер. Если заданное параметром `FieldName` поле не найдено, то метод `FindField` возвращает значение `Nil`. На практике чаще используется метод `FieldByName`, который отличается от метода `FindField` тем, что если заданное поле не найдено, то генерируется исключение.

Замечание

Имя поля, определяемое параметром `FieldName`, является именем физического поля таблицы БД, заданным при создании таблицы, а не именем (свойством `Name`) объекта `Field`, которое создано для этого поля.

Для набора данных `Query` имя `FieldName` физического поля можно переопределить в тексте SQL-запроса.

Свойство `Fields` и методы `FindField` и `FieldByName` наиболее часто используются для доступа к значению поля текущей записи совместно с такими свойствами объекта `Field`, как `AsString`, `AsInteger`, `AsFloat` или `AsBoolean`, которые позволяют обращаться к значению поля как к строковому, целочисленному, вещественному или логическому значению соответственно.

Так, в коде

```
Var x: integer;
...
Label1.Caption := Table1.FieldByName('Name').AsString;
x := Table1.FieldByName('Number').AsInteger;
```

строковое значение поля Name отображается в надписи Label1, а переменной x присваивается целочисленное значение поля Number. Если же поле Number содержит значение, которое нельзя интерпретировать как целое число, то генерируется исключение.

В рассмотренных примерах выполнялось *чтение* содержимого полей текущей записи. Аналогичным образом можно присваивать произвольному полю текущей записи *новое значение*, при этом набор данных должен находиться в режиме редактирования или вставки.

Например, с помощью следующих инструкций:

```
Var x: integer;
. . .
// Перевод набора данных в режим редактирования
Table1.Edit;
// Изменение значений полей
Table1.FieldByName('Name').AsString := Edit1.Text;
Table1.FieldByName('Number').AsInteger := x;
// Сохранение изменений
Table1.Post;
```

выполняется присвоение новых значений полям Name и Number текущей записи после того, как набор данных Table1 переведен в режим редактирования. Произведенные изменения сохраняются, а набор данных переводится в режим просмотра.

Особенности набора данных *Table*

Компонент Table представляет собой набор данных, который в текущий момент времени может быть связан только с *одной* таблицей БД. Этот набор данных формируется на базе навигационного способа доступа к данным, поэтому компонент Table рекомендуется использовать для локальных БД, таких как dBase или Paradox. При работе с удаленными БД следует использовать компонент Query. В дальнейшем при рассмотрении вопросов, связанных с локальными БД, мы обычно будем работать с компонентом Table.

Связь между таблицей и компонентом Table устанавливается через его свойство TableName типа TFileName, которое задает имя таблицы (и имя файла с данными таблицы). При задании свойства TableName указываются имя файла и расширение имени файла.

На этапе разработки приложения имена всех таблиц доступны в раскрывающемся списке Инспектора объектов. В этот список попадают таблицы, файлы которых расположены в каталоге, указанном свойством DatabaseName.

Замечание

При смене имени таблицы на этапе проектирования приложения свойство Active набора данных автоматически устанавливается в значение False. При задании имени таблицы программным способом набор данных предварительно необходимо закрыть, установив его свойство Active в значение False. В противном случае генерируется исключение.

Приведем пример, иллюстрирующий, как задается имя таблицы БД:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    Table1.Active := False;
    Table1.TableName := OpenDialog1.FileName;
    Table1.Active := True;
  end;
end;
```

Здесь нажатие кнопки `Button1` приводит к появлению диалогового окна выбора имени файла. При выборе файла таблицы его имя устанавливается в качестве значения свойства `TableName`. Набор данных `Table1` предварительно закрывается и снова открывается уже после смены таблицы. Тип таблицы определяется автоматически по расширению имени файла. При наличии ошибок, например, связанных с нарушением структуры таблицы, выдается соответствующее сообщение, а набор данных остается закрытым.

Свойство `TableType` типа `TTtableType` определяет тип таблицы. Для локальных таблиц это свойство может принимать следующие значения:

- `ttDefault` — тип таблицы автоматически определяется по расширению файла;
- `ttParadox` — таблица Paradox;
- `ttDBase` — таблица dBASE;
- `ttFoxPro` — таблица FoxPro;
- `ttASCII` — текстовый файл, содержащий данные в табличном виде (таблица ASCII).

Если свойство `TableType` имеет значение `ttDefault` (по умолчанию), то тип таблицы определяется по расширению файла:

- db или отсутствует — таблица Paradox;
- dbf — таблица dBASE;
- txt — текстовый файл (таблица ASCII).

По умолчанию в состав набора данных `Table` попадают все записи связанной с ним таблицы. Для отбора записей, удовлетворяющих определенным условиям, используются фильтры.

Delphi через BDE автоматически поддерживает многопользовательский доступ к локальным таблицам, при этом по умолчанию все пользовательские приложения имеют равные права и могут редактировать содержащиеся в таблицах данные. Чтобы запретить пользователям изменять содержание записей, можно использовать свойство `ReadOnly` типа `Boolean`. По умолчанию оно имеет значение `False`, что предоставляет пользователю право изменения записей.

Если приложению требуется получить к таблице *монопольный доступ*, то это можно осуществить через свойство `Exclusive` типа `Boolean`. По умолчанию свойство имеет значение `False`, и для таблицы, с которой связан набор данных, действует многопользовательский режим доступа. Если установить свойство `Exclusive` в значение `True`, то приложение получает монопольный доступ к соответствующей таблице, при этом другим приложениям доступ к таблице запрещается. Перед заданием значения свойства `Exclusive` набор данных должен быть закрыт, т. е. разорвана его связь с таблицей. Если

какое-либо приложение или набор данных этого же приложения уже взаимодействуют с таблицей, то попытка установить режим монопольного доступа к ней вызовет исключение.

Монопольный режим доступа необходим при выполнении таких операций, как добавление или удаление индекса методами `AddIndex` и `DeleteIndex` или очистка таблицы методом `EmptyTable`.

Замечание

Такие приложения, как Delphi и Database Desktop, также могут осуществлять доступ к таблицам. Поэтому перед отладкой приложения, которое устанавливает монопольный доступ к таблице, необходимо проверить, не работают ли с этой таблицей Delphi и/или Database Desktop. В приложении Database Desktop таблицу нужно закрыть, а в среде Delphi достаточно через Инспектор объектов установить в значение `False` свойство `Active` набора данных, связанного с таблицей.

В наборе данных `Table` можно указать текущий индекс для:

- сортировки записей;
- поиска записей;
- установления связей между таблицами.

Текущий индекс устанавливается с помощью свойства `IndexName` или `IndexFieldNames` типа `String`. На этапе разработки приложения текущий индекс выбирается в списке индексов, заданных при создании таблицы. Все возможные значения свойств `IndexName` и `IndexFieldNames` содержатся в раскрывающихся списках, доступных через Инспектор объектов. Оба свойства во многом схожи, и их использование практически одинаково. Значением свойства `IndexName` является *имя индекса*, заданное при создании таблицы, а значением свойства `IndexFieldNames` является *имя поля*, для которого был создан индекс. Если индекс состоит из нескольких полей, то для свойства `IndexName` по-прежнему задается имя этого индекса, а для свойства `IndexFieldNames` через точку с запятой перечисляются имена полей, входящие в этот индекс.

Вот как текущий индекс задается в программе:

```
Table1.IndexName := 'indName';
Table2.IndexFieldNames := 'Name';
```

Здесь компоненты `Table1` и `Table2` связаны с одной таблицей, для поля `Name` которой определен индекс `indName`. Этот индекс устанавливается в качестве текущего для обоих наборов данных.

Для таблиц Paradox сделать текущим индексом ключ (главный индекс) можно только с помощью свойства `IndexFieldNames`, перечислив ключевые поля таблицы, т. к. ключ не имеет имени и поэтому недоступен через свойство `IndexName`.

Задать ключ в качестве текущего индекса можно так:

```
Table1.IndexFieldNames := 'Name;Post;BirthDay';
```

Здесь для таблицы Paradox, с которой связан компонент `Table1`, определен ключ, в который входят поля `Name`, `Post` и `BirthDay`. Этот ключ устанавливается в качестве текущего индекса таблицы.

Замечание

Свойства IndexName и IndexFieldNames взаимозависимы. При установке значения одного из них другое автоматически очищается.

Индекс, устанавливаемый текущим, должен существовать. Если индекс, задаваемый как значение свойства IndexName или IndexFieldNames, для таблицы не существует, то возникает исключение.

При смене таблицы, с которой ассоциирован компонент Table, значения свойств IndexName и IndexFieldNames не изменяются автоматически, поэтому программист должен установить нужные значения самостоятельно.

Получить доступ к полям в составе текущего индекса можно с помощью свойств IndexFieldCount и IndexFields.

Свойство IndexFieldCount типа Integer содержит число полей в текущем индексе и доступно для чтения при выполнении приложения.

Свойство IndexFields[Index: Integer] типа TField позволяет обращаться к полям текущего индекса, при этом переменная Index задает номер индекса в массиве полей этого индекса (отсчет начинается с нуля). Класс TField представляет собой поле набора данных и имеет большое число свойств и методов.

Чаще всего индексы определяются при создании таблицы и в дальнейшем при работе с таблицей не изменяются. Однако у программиста есть возможность изменять определенные для таблицы индексы динамически, т. е. в процессе выполнения приложения, с помощью методов AddIndex и DeleteIndex.

Замечание

Допускается изменять индексы для таблицы, открытой в режиме монопольного доступа, когда свойство Exclusive имеет значение True.

Процедура AddIndex(const Name, Fields: String; Options: TIndexOptions) добавляет к таблице индекс, имя которого задано параметром Name. Входящие в состав индекса поля указываются в параметре Fields; если индекс состоит из нескольких полей, то они разделяются точкой с запятой. Указывать можно только поля, которые входят в структуру таблицы, в противном случае генерируется исключение, а индекс не создается. Параметр Options содержит параметры индекса. Он имеет множественный тип и может принимать комбинации следующих значений:

- ixPrimary (первичный индекс);
- ixUnique (уникальный индекс) — для этого индекса не допускается повторение значений полей в его составе;
- ixDescending (сортировка в порядке убывания значений) — по умолчанию строится индекс, определяющий сортировку по возрастанию;
- ixExpression (индекс создается на основе выражения) — только для таблиц dBase;
- ixCaseInsensitive (сортировка не зависит от регистра букв);
- IxNonMaintained (индекс автоматически не изменяется, если таблица открыта).

Приведем пример, демонстрирующий добавление индекса к таблице:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  // Перевод таблицы в режим монопольного доступа
  Table1.Close;
  Table1.Exclusive := True;
  Table1.Open;
  // Добавление индекса
  Table1.AddIndex('indPosition', 'Position',
                  [ixDescending, ixCaseInsensitive]);
  // Закрытие режима монопольного доступа к таблице
  Table1.Close;
  Table1.Exclusive := False;
  Table1.Open;
end;
```

Здесь к таблице, с которой связан набор данных `Table1`, добавляется индекс `indPosition`, построенный по полю `Position`. Для нового индекса определяется порядок сортировки по убыванию значений, не зависящий от регистра букв.

Процедура `DeleteIndex(const Name: String)` удаляет из таблицы индекс, имя которого задано параметром `Name`. При попытке удаления несуществующего индекса генерируется исключение.

В программе удаление индекса из таблицы можно реализовать так:

```
Table1.DeleteIndex('indPosition');
```

Перед удалением индекса таблица должна быть переведена в режим монопольного доступа аналогично тому, как это происходило при добавлении индекса.

Для доступа к информации обо *всех* индексах, определенных для таблицы и, соответственно, для связанного с ней набора данных, можно использовать свойство `IndexDefs` типа `TIndexDefs`. Это свойство доступно как на этапе разработки, так и на этапе выполнения приложения. Класс `TIndexDefs` в свою очередь также имеет полезные свойства и методы, которые мы сейчас и рассмотрим.

Перед работой с индексами всегда рекомендуется вызывать метод `UpDate`, который производит обновление информации об индексах так, чтобы значение свойства `IndexDefs` отражало реальное состояние с учетом сделанных изменений.

Свойство `Count` типа `Integer` содержит количество индексов и доступно для чтения. Управление количеством индексов осуществляется косвенно, т. е. через другие свойства и методы.

Свойство `Items[Index: Integer]` типа `TIndexDef` представляет собой список, содержащий информацию обо всех индексах таблицы. Переменная `Index` указывает номер индекса в списке, отсчет начинается с нуля. Тип `TIndexDef` (не путать с классом `TIndexDefs`) является классом и, в свою очередь, также имеет свойства, главные из которых — `Name`, `Fields` и `Options`.

Замечание

При доступе к информации об индексах можно не указывать свойство `Items`. Например, чтобы считать в редактор `Edit1` список полей, составляющих второй индекс, можно использовать следующие эквивалентные по действию инструкции:

```
Edit1.Text := Table1.IndexDefs[1].Fields;
```

или

```
Edit1.Text := Table1.IndexDefs.Items[1].Fields;
```

Свойство `Name` типа `String` содержит имя индекса. Для индексов, построенных на основе выражений (таблицы dBase), и для главного индекса таблиц Paradox вместо имени возвращается пустая строка.

Свойство `Fields` типа `String` содержит имена полей, по которым построен индекс. Если в индексе используется несколько полей, то их имена разделяются точкой с запятой.

Свойство `Options` типа `TIndexOption` содержит параметры индекса, заданные при его создании.

Рассмотрим работу со списком индексов и индексных полей на примере.

В форме расположены следующие компоненты: сетка `DBGrid`, в которой отображаются записи таблицы БД, два списка `ListBox` с соответствующими надписями, а также две кнопки `Button`. При нажатии кнопки **Индексы** (`Button1`) в список `ListBox1` загружается список индексов, определенных для набора данных `Table1`, а в список `ListBox2` — список полей, образующих эти индексы (рис. 17.2). Так как компонент `Table1` связан с таблицей Paradox, то имя главного индекса, построенного по полю `P_Code`, содержит пустую строку.

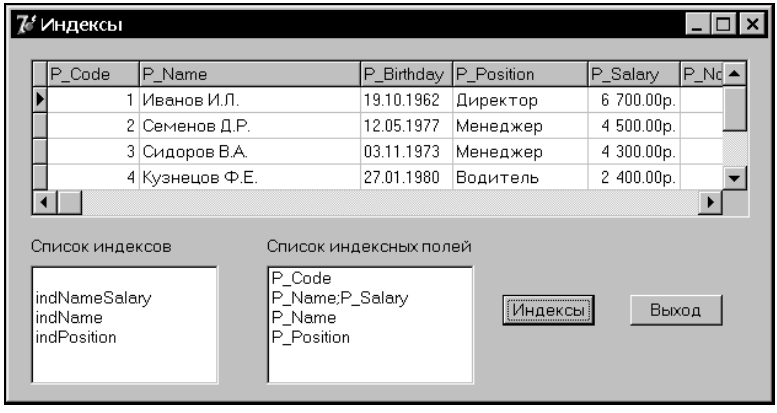


Рис. 17.2. Получение списка индексов и полей

Обработчик события нажатия кнопки `Button1` выглядит так:

```
procedure TForm1.Button1Click(Sender: TObject);
var n: integer;
begin
  for n := 0 to Table1.IndexDefs.Count – 1 do begin
    ListBox1.Items.Add(Table1.IndexDefs[n].Name);
```

```
    ListBox2.Items.Add(Table1.IndexDefs[n].Fields);
  end;
end;
```

Для получения списка имен индексов можно использовать метод `GetItemNames(List: TStrings)`. Он возвращает список имен индексов через параметр `List`, в качестве которого можно использовать любой объект типа `TStrings`, например, `ListBoxItems`.

Получение списка имен индексов в программе может выглядеть так:

```
ListBox1.Items.Clear;
Table1.GetItemNames(ListBox1.Items);
```

Для добавления и удаления индексов используются методы `Add` и `Clear`. Процедура `Add(const Name, Fields: String; Options: TIndexOptions)` добавляет новый индекс. Параметры этой процедуры не отличаются от параметров метода `AddIndex` набора данных `Table`, однако для метода `Add` не требуется перевод таблицы в режим монопольного доступа, что делает его использование более удобным.

Замечание

Для таблиц Paradox при определении ключа (главного индекса) имя индекса не задается, т. к. этот индекс не именуется.

Процедура `Clear` удаляет все индексы, содержащиеся в списке индексов.

Вот как удаление и добавление индексов к таблице реализуется в программе:

```
Table1.IndexDefs.Clear;
Table1.IndexDefs.Add('', 'Number', [ixPrimary, ixUnique]);
Table1.IndexDefs.Add('indMain', 'Code;Position',
   [ixDescending, ixCaseInsensitive]);
```

В этом примере из набора данных `Table1`, связанного с таблицей Paradox, удаляются все индексы и создаются два новых индекса: первичный индекс, построенный по полю `Number`, и вторичный индекс `indMain`, построенный по полям `Code` и `Position`.

Ряд методов предназначен для поиска индексов в списке, например, `IndexOf`, `Find` и `FindIndexForFields`. Их удобно использовать в случае, когда индексы или их поля вводит пользователь на этапе выполнения приложения. Перед использованием таких индексов целесообразно проверять, существуют ли они в списке индексов, т. к. в противном случае мы рискуем получить исключение.

Функция `IndexOf(const AName: String): Integer` осуществляет поиск индекса по имени, заданному параметром `AName`. В качестве результата возвращается номер индекса в свойстве `Items`. В случае неудачного поиска возвращается значение `-1`.

Рассмотрим пример поиска индекса:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if Table1.IndexDefs.IndexOf(Edit1.Text) = -1 then begin
    MessageDlg('Индекс отсутствует!', mtError, [mbOK], 0);
    if Edit1.CanFocus then Edit1.SetFocus;
  end;
end;
```

Здесь при нажатии кнопки Button2 выполняется проверка существования индекса, имя которого введено в поле ввода Edit1. При отсутствии в наборе данных Table1 указанного индекса выдается сообщение об ошибке, и фокус ввода устанавливается на Edit1.

Функция FindIndexForFields(const Fields: String): TIndexDef осуществляет поиск индекса по списку полей, заданному параметром Fields. В случае успешного поиска функция возвращает информацию об индексе, если же индекс отсутствует, то генерируется исключение.

Замечание

Если не найден индекс, поля которого полностью совпадают с заданными, но есть индексы, которые включают в себя все заданные поля, то в качестве результата возвращается первый такой индекс.

При запуске приложения информация об индексах таблицы считывается с диска из соответствующих индексных файлов. При динамическом, т. е. в процессе выполнения приложения, изменении индексов содержимое индексных файлов остается прежним. При необходимости программист должен предусмотреть сохранение новой информации на диске для ее последующего применения. Для этого можно использовать свойство StoreDefs типа Boolean, которое указывает, нужно ли сохранять информацию об индексах на диске. При любых изменениях в индексах это свойство нужно установить в значение True, тогда информация об индексах будет автоматически сохранена на диске. Если изменения в индексах сохранять не нужно, то достаточно установить свойство StoreDefs в значение False, и изменения в индексах будут действовать только при выполнении приложения.

Особенности набора данных *Query*

Компонент Query представляет собой набор данных, записи которого формируются в результате выполнения SQL-запроса и основаны на реляционном способе доступа к данным. При работе с удаленными БД рекомендуется использовать именно набор данных Query.

Замечание

При работе с удаленными БД следует обращаться к средствам языка SQL. Это относится и к таким операциям, как перемещение по набору данных или вставка в него записей. Если же для компонента Query используются методы типа Next или Insert, то вместо реляционного способа доступа к удаленным данным будет применен навигационный. В результате набор данных Query будет мало чем отличаться от набора данных Table.

В отличие от компонента Table, набор данных Query может включать в себя записи более чем одной таблицы БД.

Текст запроса, на основании которого в набор данных отбираются записи, содержится в свойстве SQL типа TStrings. Запрос включает в себя команды на языке SQL и выполняется при открытии набора данных. Запрос SQL иногда называют SQL-программой.

При формировании запроса на этапе разработки приложения можно использовать текстовый редактор (рис. 17.3), вызываемый через Инспектор объектов двойным щелчком в области значения свойства SQL.

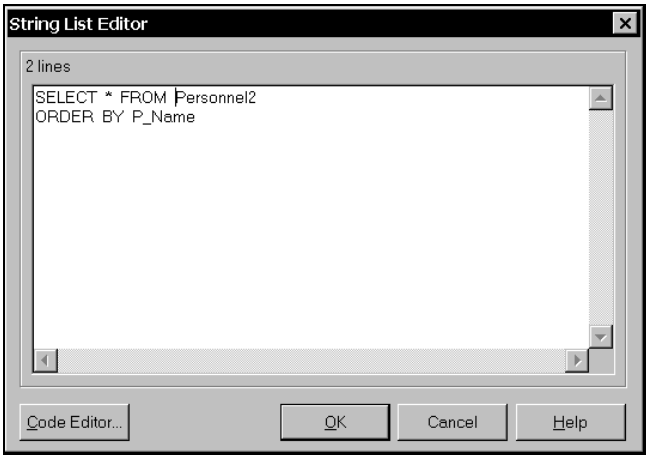


Рис. 17.3. Редактирование запроса SQL

SQL-запрос также можно формировать и изменять динамически, внося изменения в его текст (значение свойства `SQL` компонента `Query`) непосредственно при выполнении приложения.

Замечание

В процессе формирования SQL-запроса проверка его правильности не производится, и если в запросе имеются ошибки, то они выявляются только при открытии набора данных. Одним из вариантов предотвращения ошибок в SQL-запросе является его предварительная отладка, например, с помощью программы Database Desktop.

Рассмотрим пример приложения — простейшего редактора, позволяющего подготавливать и выполнять SQL-запросы. На рис. 17.4 показана форма приложения при его выполнении. Кроме визуальных компонентов, форма содержит два компонента доступа к данным — `Query1` и `DataSource1`, которые при выполнении приложения не видны.

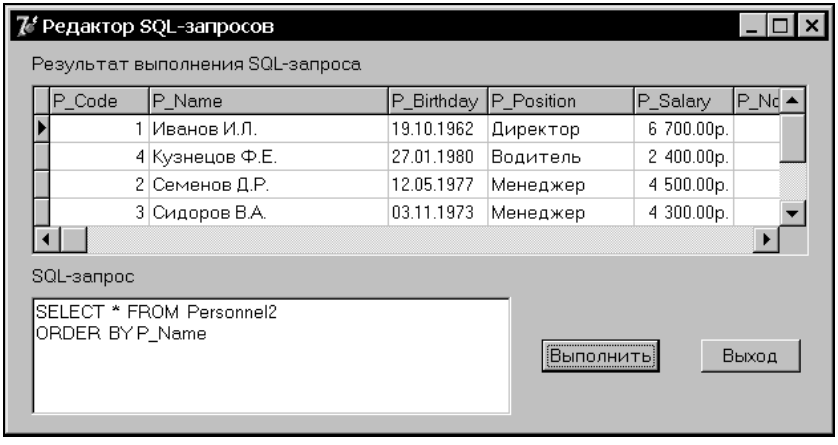


Рис. 17.4. Приложение-редактор SQL-запросов

Редактирование SQL-запроса осуществляется с помощью компонента `Memo1`. Набранный запрос выполняется при нажатии кнопки **Выполнить** (`Button1`), а результат выполнения отображается в компоненте `DBGrid1`.

При наличии в тексте SQL-запроса ошибки генерируется исключение и выдается сообщение об ошибке (рис. 17.5), а результат запроса не определен. При этом набор данных `Query1` автоматически закрывается.

Значения свойств `DataSet` источников данных `DataSource1` и `DataSource` компонента `DBGrid1`, с помощью которых организуется взаимодействие компонентов `Query1`, `DataSource1` и `DBGrid1`, устанавливаются при создании формы. В последующих примерах приложений значения этих свойств задаются через Инспектор объектов, поэтому инструкции, присваивающие свойствам необходимые значения, в модуле формы отсутствуют.

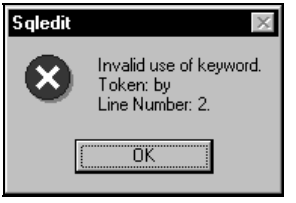


Рис. 17.5. Сообщение об ошибке в тексте SQL-запроса

В листинге 17.1 приведен код модуля `uSQLEdit` формы `Form1` приложения.

Листинг 17.1. Пример редактора SQL-запросов

```
unit uSQLEdit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Grids, DBGrids, Db, DBTables;
type
  TForm1 = class(TForm)
          Memo1: TMemo;
    DataSource1: TDataSource;
         Query1: TQuery;
        DBGrid1: TDBGrid;
        Button1: TButton;
         Label1: TLabel;
         Label2: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
```

```
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  DataSource1.DataSet := Query1;
  DBGrid1.DataSource := DataSource1;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.SQL.Assign(Memo1.Lines);
  Query1.Open;
end;
end.
```

Метод `Assign` выполняет присваивание одного объекта другому, при этом объекты должны иметь совместимые типы. Применительно к списку строк (класс `TStrings`), которому принадлежат свойства SQL компонента `Query1` и `Lines` компонента `Memo1`, подобное присваивание означает копирование информации из одного списка в другой с заменой содержимого последнего. Если размеры списков (число элементов) не совпадают, то после замены число элементов заменяемого списка становится равным числу элементов копируемого списка.

Компонент `Query` обеспечивает выполнение SQL-запроса и является набором данных, который формируется на основе этого запроса. Формирование набора данных выполняется при активизации компонента `Query` вызовом метода `Open` или установкой свойства `Active` в значение `True`. В ряде случаев при выполнении SQL-запроса не требуется возвращать набор данных, например, при удалении, вставке или изменении записей. В этом случае предпочтительнее выполнять запрос компонента `Query` не его открытием, а вызовом метода `ExecSQL`. При работе в сети вызов метода `ExecSQL` выполняет требуемую модификацию набора данных, не передавая в вызывающее приложение (компьютер) записи набора данных, что заметно снижает нагрузку на сеть.

Компонент `Query` может быть связан с таблицей БД или напрямую, или содержать *копии* отобранных записей таблицы, доступные для чтения. Вид взаимодействия определяется свойством `RequestLive` типа `Boolean`. По умолчанию свойство имеет значение `False`, и набор данных `Query` доступен только для чтения. Если пользователю или программисту требуется возможность редактирования записей, то свойство `RequestLive` нужно установить в значение `True`. В этом случае набор данных `Query` напрямую связывается с соответствующей таблицей, аналогично набору данных `Table`.

Замечание

Влияние свойства `RequestLive` зависит от текста выполняемого SQL-запроса. Если в результате выполнения запроса не может быть получен редактируемый набор данных, то установка свойства `RequestLive` в значение `True` игнорируется.

Чтобы проверить результат установки значения свойства `RequestLive`, можно воспользоваться свойством `CanModify` типа `Boolean`. Если оно имеет значение `True`, то набор данных является редактируемым, если `False` — то нередактируемым.

Чтобы получить в результате выполнения SQL-запроса редактируемый набор данных, кроме установки свойства RequestLive в значение True, должны быть выполнены следующие условия:

- данные отбираются только из одной таблицы или просмотра (view);
- таблица или просмотр допускают модификацию;
- в запросе не используется операнд DISTINCT и агрегатные (статистические) функции;
- в запросе не применяется соединение таблиц;
- в запросе отсутствуют подзапросы и вложенные запросы;
- не используется группирование данных;
- сортировка применяется только к индексированным полям.

Для локальных БД вместо компонента Table можно также использовать компонент Query. Если установить свойство SQL в значение SELECT * FROM NameTableBD, а свойство RequestLive — в значение True, то набор данных Query будет аналогичен набору данных Table. (Здесь NameTableBD является именем таблицы БД, которое для компонента Table задается в свойстве TableName.) Однако набор данных Query, в отличие от Table, не имеет системы индексов, поэтому к Query неприменимы методы, использующие индексирование, например, методы FindFirst, FindLast, FindNext и FindPrior.

Более подробно компонент Query и язык SQL рассматриваются в *главе 20*, посвященной реляционному доступу к данным.

Объекты поля

Объект поля Field имеет тип TField и служит полем набора данных. Тип TField является абстрактным классом и непосредственно не используется. Вместо него применяются производные классы (табл. 17.1), соответствующие типу данных, размещаемых в рассматриваемом поле набора данных. Производные классы отличаются от базового класса TField некоторыми особенностями, связанными с манипулированием конкретным типом данных, например, символьным, числовым или логическим. Далее под объектами типа TField мы будем понимать либо сам объект типа TField, либо один из производных от него объектов, например, типа TStringField (строковое значение) или TIntegerField (целочисленное значение).

Таблица 17.1. *Типы объектов* Field

Тип объекта	Вид поля
TBLOBField	Поле BLOB-значения (BLOB, Binary Large Object — большой двоичный объект)
TMemoField	Мемо-поле (поле комментария)
TGraphicField	Графическое поле
TBooleanField	Поле логического значения
TBinaryField	Поле двоичного значения
TBytesField	Поле байтового значения фиксированной длины

Таблица 17.1 *(окончание)*

Тип объекта	Вид поля
TVarBytesField	Поле байтового значения переменной длины
TDateTimeField	Поле значения даты и времени
TDateField	Поле значения даты
TTimeField	Поле значения времени
TNumericField	Поле числового значения
TBCDField	Поле BCD-значения
TFloatField	Поле вещественного значения
TCurrencyField	Поле значения денежной суммы
TIntegerField	Поле целочисленного значения (32 разряда)
TAutoincField	Поле автоинкрементного значения (32 разряда)
TSmallintField	Поле целочисленного короткого значения (16 разрядов)
TLargeintField	Поле целочисленного длинного значения (64 разряда)
TWordField	Поле целочисленного значения без знака (16 разрядов)
TStringField	Поле строкового значения

Объекты типа TField являются невизуальными и служат для доступа к данным соответствующих полей записей. Управляя объектами типа TField, можно управлять поведением полей, при этом все объекты полей являются независимыми друг от друга. Например, разработчик может запретить изменять значение отдельного поля, несмотря на то что набор данных в целом является модифицируемым и допускает изменение значений других полей. Кроме того, можно скрыть то или иное поле от пользователя, сделав его невидимым.

Задать состав полей набора данных можно двумя способами:

- по умолчанию (динамические поля);
- с помощью редактора полей (статические поля).

По умолчанию при каждом открытии набора данных как на этапе проектирования, так и на этапе выполнения приложения для каждого поля набора автоматически создается свой объект типа TField. В этом случае мы имеем дело с *динамическими полями*, достоинством которых является корректность отображения структуры набора данных даже при ее изменении. Напомним, что для компонента Table состав полей определяется структурой таблицы, с которой этот компонент связан, а для компонента Query состав полей зависит от SQL-запроса.

Однако у использования динамических полей есть и существенные недостатки, связанные с тем, что для полученного набора данных нельзя выполнить такие действия, как ограничение состава полей или определение вычисляемых полей. Поэтому при необходимости этих операций следует использовать второй способ задания состава полей.

На этапе разработки приложения с помощью Редактора полей можно создавать для набора данных *статические* (устойчивые) поля, основные достоинства которых состоят в реализации следующих возможностей:

- определение вычисляемых полей, значения которых рассчитываются с помощью выражений, использующих значения других полей;
- ограничение состава полей набора данных;
- изменение порядка полей набора данных;
- скрытие или показ отдельных полей при выполнении приложения;
- задание формата отображения или редактирования данных поля на этапе разработки приложения.

Необходимо отметить, что при модификации структуры таблицы, например, удалении поля или изменении его типа, открытие набора данных, имеющего статические поля, может привести к возникновению исключения.

Редактор полей

По умолчанию для каждого физического поля при открытии набора данных автоматически создается объект типа `TField`, а все поля в наборе данных являются динамическими и доступными. Для создания статических (устойчивых) полей используется специальный Редактор полей. В случае, если хотя бы одно поле набора данных является статическим, динамические поля больше создаваться не будут. Таким образом, в наборе данных будут доступны только статические поля, а все остальные считаются отсутствующими. Определить или отменить состав статических полей можно с помощью Редактора полей на этапе разработки приложения.

Замечание

Для компонента `Query` состав полей определяется также в тексте SQL-запроса, с помощью которого можно задать или изменить состав полей набора данных, несмотря на то что эти поля являются динамическими.

Во время выполнения приложения можно определить вид полей набора данных с помощью свойства `DefaultFields` типа `Boolean`. Если его значение равно `True`, то набор данных имеет поля по умолчанию, т. е. динамические, в противном случае для набора данных заданы статические поля.

Для запуска Редактора полей (рис. 17.6) следует сделать двойной щелчок на компоненте `Table` или `Query` или вызвать для этих компонентов контекстное меню правой кнопкой мыши и выбрать пункт **Fields Editor**. В заголовке Редактора полей выводится составное имя набора данных, например, `Form1.Table1`. Для перемещения по полям используются четыре кнопки навигации Редактора или мышь. Бóльшую часть Редактора занимает список статических полей, при этом поля перечисляются в порядке их создания, который может отличаться от порядка полей в таблице БД.

Первоначально список статических полей пуст, указывая, что все поля набора данных являются динамическими. С помощью Редактора полей разработчик может выполнить следующие операции:

- создать новое статическое поле;
- удалить статическое поле;
- изменить порядок следования статических полей.

Кроме того, для любого выбранного в редакторе статического поля с помощью Инспектора объектов можно задать или изменить свойства этого поля (объекта типа `TField`) и определить обработчики его событий. Подобные действия разрешается производить благодаря тому, что соответствующие статическим полям объекты типа `TField` доступны на этапе разработки приложения.

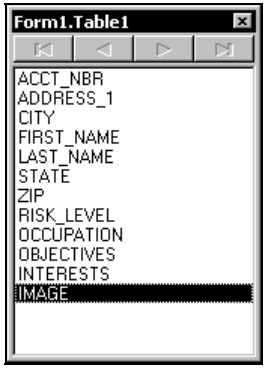


Рис. 17.6. Редактор полей

Рис. 17.7. Добавление новых статических полей

Для создания статического поля следует вызвать контекстное меню Редактора полей и выбрать пункт **Add Fields** (Добавить поля), в результате чего появляется диалоговое окно добавления новых полей (рис. 17.7). В списке **Available fields** (Доступные поля) окна содержатся все те поля набора данных, которые еще не являются статическими. После выбора одного или нескольких полей и нажатия кнопки **OK** эти поля добавляются в состав статических полей набора данных. Добавленное статическое поле является полем *данных* и связано с конкретным физическим полем таблицы БД.

Для добавления в список всех физических полей таблицы (для набора данных `Table`) или результата выполнения SQL-запроса (для набора данных `Query`) нужно выбрать в контекстном меню Редактора полей пункт **Add all Fields** (Добавить все поля).

При каждом открытии (в том числе при разработке приложения) набора данных проверяется возможность создания экземпляров класса `TField` для статических полей. Если поле вследствие какой-нибудь ошибки является недопустимым, и создание объекта типа `TField` невозможно, то генерируется исключение, а набор данных закрывается.

Для удаления статического поля нужно выбрать пункт **Delete** контекстного меню или выделить поле в списке и нажать клавишу <Delete>. После удаления статического поля оно становится недоступным для операций в программе, однако в случае необходимости его снова можно сделать статическим, добавив в список Редактора полей. При этом следует иметь в виду, что все свойства этого поля устанавливаются заново, а все сделанные ранее изменения теряются.

Замечание

Если удалены все статические поля, то все поля набора данных становятся динамическими и доступными при выполнении приложения.

Порядок следования полей определяется их местом в списке Редактора полей. По умолчанию порядок полей соответствует порядку физических полей в таблицах БД. Его можно изменить, перемещая поля в списке с помощью мыши или комбинаций клавиш <Ctrl>+<Page Up> и <Ctrl>+<Page Down>.

Замечание

Если для набора данных определены статические поля, то изменение значения свойства `TableName` этого набора данных может привести к ошибке, что обычно и происходит. Это связано с тем, что в новой таблице, связываемой с набором данных, могут отсутствовать физические поля, для которых были созданы статические поля. В таких случаях программист должен предусматривать соответствующие операции, например, формирование нового состава статических полей.

Есть три типа статических полей:

- поле данных, связанное с соответствующим физическим полем таблицы;
- вычисляемое поле, значение которого рассчитывается в обработчике события `OnCalcFields` во время выполнения приложения;
- поле выбора, значение которого можно выбирать из списка, формируемого на основе заданных критериев и правил.

Для создания нового статического поля любого типа нужно выбрать в контекстном меню Редактора полей пункт **New Field**, в результате чего открывается одноименное диалоговое окно (рис. 17.8).

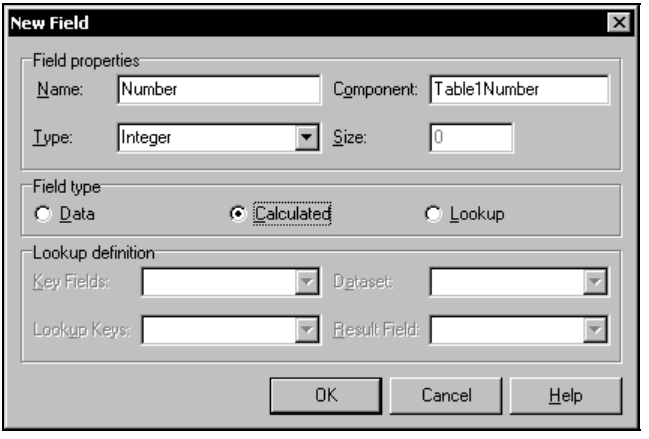


Рис. 17.8. Окно создания статического поля

Для задания общих свойств (параметров) нового поля используется группа элементов управления **Field properties** (Свойства поля). В поле ввода **Name** задается значение свойства `FieldName` (имя поля), а в поле ввода **Component** — значение свойства `Name` (имя компонента поля — объекта типа `TField`). При программировании обычно ис-

пользуется имя поля. Delphi автоматически формирует значение в поле ввода **Component**, и попытка изменить его не приводит к желаемому результату. В списке **Type** и поле ввода **Size** указываются тип данных и размер поля. Тип данных обязательно задается для всех полей, а необходимость задания размера зависит от типа данных. Например, для поля с типом данных `Integer` задание размера не имеет смысла, а для типа `String` размер поля ограничивает максимальную длину строки.

Тип нового поля выбирается в группе переключателей **Field type** из следующих вариантов:

- **Data** (поле данных);
- **Calculated** (вычисляемое поле);
- **Lookup** (поле выбора).

В группе **Lookup definition** (Определение выбора) для поля выбора устанавливаются такие параметры, как набор данных и поля связи, а также поля для формирования списка выбора и результата.

После создания нового статического поля его свойства становятся доступными через Инспектор объектов и могут быть изменены. При этом каждому параметру, задаваемому с помощью поля ввода или переключателя окна **New Field** (см. рис. 17.8), соответствует определенное свойство объекта типа `TField` (табл. 17.2). Все свойства объекта доступны через Инспектор объектов, за исключением свойства `FieldName`, которое доступно только во время выполнения приложения, однако при разработке приложения значение этого свойства видно в окне Редактора полей. Значение параметра `Type` определяет класс объекта `Field`, который будет соответствовать статическому полю, например, для типа `String` это `TStringField`, а для типа `Float` — это `TFloatField`.

Таблица 17.2. *Свойства объекта поля*

Свойство объекта поля	Элемент управления
`FieldName`	Поле ввода **Name**
`Name`	Поле ввода **Component**
`Size` (для строковых полей), `Precision` (для числовых полей)	Поле ввода **Size**
`FieldKind`	Группа переключателей **Field type**
`KeyFields`	Комбинированный список **Key Fields**
`LookupKeyFields`	Комбинированный список **Lookup Keys**
`LookupDataset`	Комбинированный список **Dataset**
`LookupResultField`	Комбинированный список **Result Field**

Свойство `FieldKind` типа `TFieldKind` определяет тип поля и принимает следующие значения:

- `fkData` (поле данных);
- `fkCalculated` (вычисляемое поле);

- `fkLookupField` (поле выбора);
- `fkInternalCalc` (вычисляемое поле, которое сохраняется в наборе данных);
- `fkAggregate` (поле, содержащее агрегированный результат).

Статическое поле данных создается для соответствующего физического поля таблицы рассмотренным выше способом, в то время как создание вычисляемого поля или поля выбора является более сложной задачей.

Для создания статического вычисляемого поля нужно выполнить следующие действия:

1. В окне создания статического поля задать имя и тип поля, а также выбрать переключатель **Calculated**.
2. Для набора данных, содержащего поле, подготовить код обработчика события `OnCalcFields` типа `TDataSetNotifyEvent`, в котором этому полю присваивается требуемое значение, при этом для расчета значения вычисляемого поля можно использовать значения других полей, а также переменные и константы программы.

После создания вычисляемого поля его свойство `FieldKind` автоматически получает значение `fkCalculated`. Кроме того, также автоматически свойство `Calculated` типа `Boolean` устанавливается в значение `True`, указывающее на то, что это поле является вычисляемым. Для полей другого типа свойство `Calculated` имеет значение `False`.

Событие `OnCalcFields` предназначено для определения значений всех вычисляемых полей набора данных. Оно генерируется каждый раз при считывании записи из таблицы, а также при изменении значений невычисляемых полей, если свойство `AutoCalcFields` типа `Boolean` установлено в значение `True` (по умолчанию). На время выполнения обработчика события `OnCalcFields` набор данных переводится в режим `dsCalcFields` расчета вычисляемых полей, а затем возвращается в предыдущий режим.

Замечание

В обработчике события `OnCalcFields` можно присваивать значения вычисляемым полям, а также полям выбора (в том числе не входящим в состав списка выбора). Попытка изменить значение поля данных (физического поля таблицы) вызывает исключение.

Рассмотрим пример, иллюстрирующий использование вычисляемых полей.

Пусть в состав таблицы входят три поля: `Name` (название товара), `Price` (цена единицы товара, в рублях) и `Number` (количество товара). Набор данных должен содержать для каждого товара также данные об общей стоимости товара в рублях и в евро. Общая стоимость определяется на основании цены единицы товара и количества товара, поэтому в набор данных добавляются два вычисляемых поля: `Total` и `TotalEvro` для рублей и евро соответственно (рис. 17.9).

Для вычисления значений новых полей используется код обработчика события `OnCalcFields`, приведенный далее. Обращение к вычисляемым полям и полям данных выполнено разными способами: по имени компонента (для вычисляемых полей `Total` и `TotalEvro`) и по имени поля (для полей данных `Price` и `Number`). Следует иметь в виду, что поля `Price` и `Number` тоже должны быть статическими, в противном случае при попытке обращения к ним произойдет ошибка. В примере статическим является также поле `Name`.

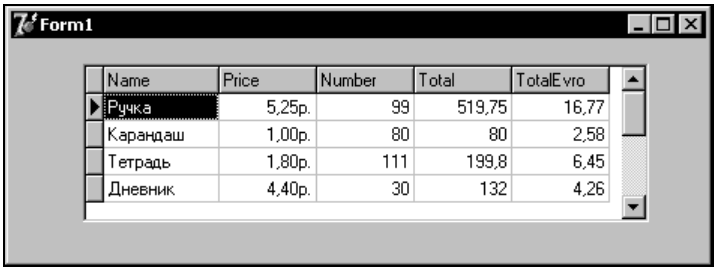


Рис. 17.9. Использование вычисляемых полей

При расчете стоимости в условных единицах предполагается, что обменный курс составляет 31. В принципе значение этого коэффициента должно быть переменным значением с возможностью его редактирования пользователем.

```
// Отформатировать значения поля можно следующим образом:
// Table1TotalEvro.DisplayFormat:='#####.##'
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
  Table1Total.AsFloat := Table1.FieldByName('Price').AsFloat*
                         Table1.FieldByName('Number').AsInteger;
  Table1TotalEvro.AsFloat := Table1Total.AsFloat/31;
end;
```

Если в качестве обменного курса взять некоторое реальное значение, например, `31.75`, то значения поля `TotalEvro` будут содержать большое количество дробных разрядов. Поэтому необходимо округлять получаемые значения или выводить полученные значения в требуемом формате. Для форматирования значений можно использовать свойство `DisplayFormat` этого поля, установив его через Инспектор объектов или при создании формы, например, следующей инструкцией:

```
Table1TotalEvro.DisplayFormat := '#####.##';
```

Поле выбора предоставляет возможность выбора одного значения из предлагаемого списка и автоматического занесения информации в заданное поле модифицируемой записи. С полем выбора связывается специальный список, заполняемый значениями указанного поля из второго набора данных. Оба набора данных связываются по полю (полям) связи.

Необходимость использования поля выбора может возникнуть в случае, когда при поступлении товара ведется таблица реестра, в которую заносится такая, например, информация (в скобках приведены названия полей таблицы):

- уникальный номер записи (`R_N`);
- дата поступления товара (`R_Date`);
- уникальный код товара (`R_Code`);
- количество поступившего товара (`R_Number`);
- примечание (`R_Note`).

Данные о каждом конкретном товаре хранятся в другой таблице, имеющей поля:

- уникальный код товара (G_Code);
- название товара (G_Name);
- единица измерения товара (G_Unit);
- цена единицы товара (G_Price);
- примечание (G_Note).

В таблице данных о товаре поле уникального кода товара (G_Code) является автоинкрементным — его значение формируется автоматически при добавлении нового товара. При добавлении новой записи в таблицу реестра в поле кода товара (R_Code) должен заноситься код поступившего товара. Для установления соответствия между поступившими товарами и их характеристиками обе таблицы связываются по этим полям. Однако при занесении нового товара в реестр пользователь должен самостоятельно задавать код, соответствующий поступившему товару. Эта операция неудобна для пользователя, т. к. нужно знать эти коды, и может привести к ошибкам.

Более удобным вариантом в данном случае будет формирование поля выбора, содержащего товары, из которого пользователь выбирает одно из наименований. Тогда после выбора товара соответствующий код будет заноситься в поле R_Code таблицы реестра автоматически. Окно создания поля выбора показано на рис. 17.10.

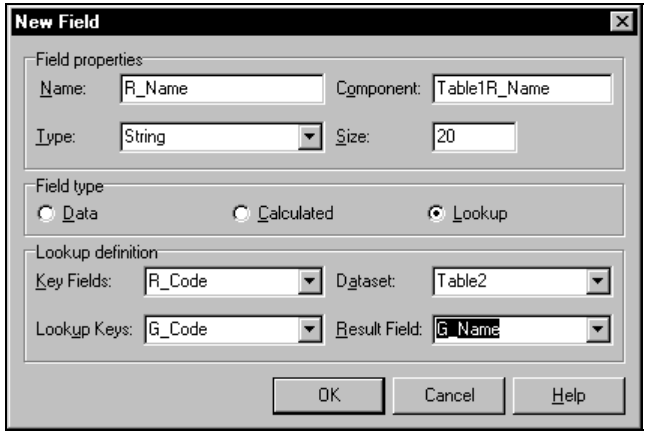


Рис. 17.10. Окно создания поля выбора

Поле выбора названо R_Name и принадлежит набору данных Table1, содержащему записи таблицы реестра. Для формирования списка выбора используется поле G_Name набора данных Table2, содержащего записи таблицы данных товара. Связь между двумя наборами данных Table1 и Table2 осуществляется через их поля кода товара R_Code и G_Code соответственно (рис. 17.11).

Использование поля выбора заключается в том, что пользователь выбирает значение в поле R_Name, содержащем список, который построен на основании значений поля G_Name. После выбора значения для поля R_Name из поля связи G_Code автоматически заносится соответствующее значение в поле R_Code. Таким образом, поле R_Name, со-

держащее список, используется для выбора, а поле связи `R_Code` — для занесения в него значения.

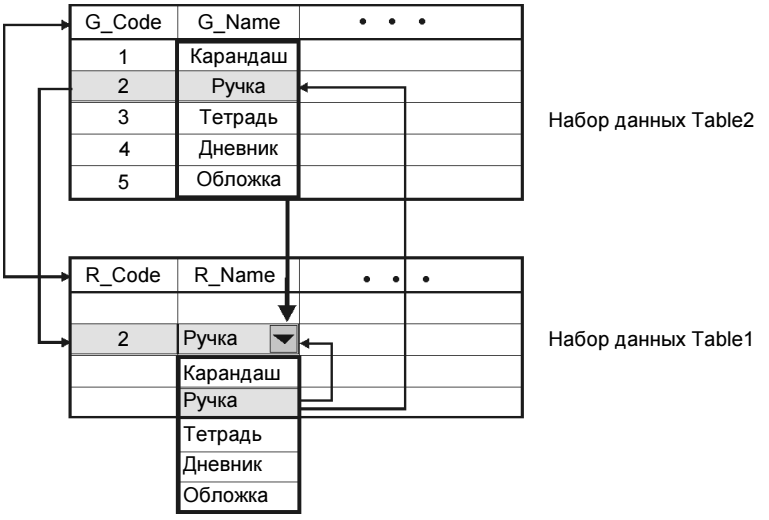


Рис. 17.11. Схема использования поля выбора

Замечание

Поля связи не обязательно должны быть индексными.

В частном случае поле выбора и поле связи могут быть одним и тем же полем. Это может потребоваться, например, в случае, когда нет необходимости автоматически вводить код для выбранного значения, а достаточно просто выбрать значение в списке.

На рис. 17.12 демонстрируется использование поля выбора. В верхней части формы отображается таблица реестра, в нижней — таблица данных, которая приведена для

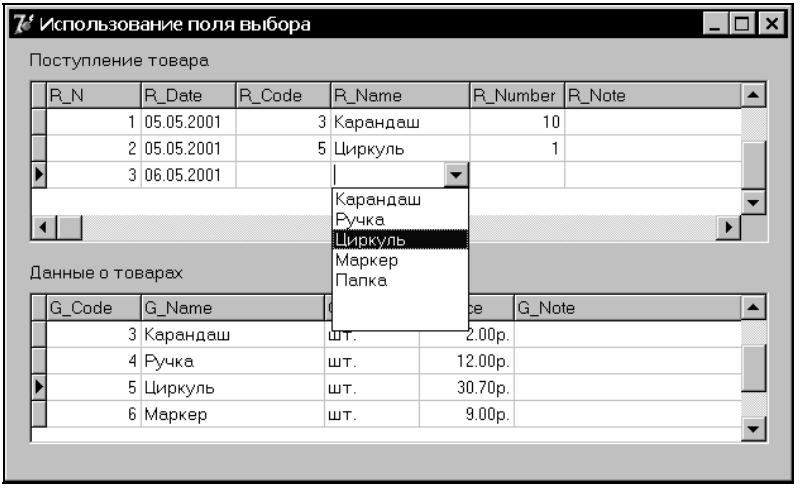


Рис. 17.12. Использование поля выбора

наглядности. На практике связанный набор данных обычно не показывается. При выборе значения в списке поля выбора для связанного набора данных текущий указатель устанавливается на запись, значение из которой было выбрано.

При изменении набора данных, по полю которого построен список выбора, автоматического изменения списка не происходит. Обновление списка выбора является обязанностью программиста, его удобно выполнять с помощью метода `RefreshLookupList`.

Операции с полями

Через объект типа `TField` разработчик может:

- обратиться к полю и его значению;
- проверить тип и значение поля;
- отформатировать значение поля, отображаемое или редактируемое в визуальных компонентах.

При этом динамические и статические поля имеют одинаковые свойства, события и методы, с помощью которых можно управлять этими объектами при выполнении приложения. В связи с тем, что статические поля определяются на этапе разработки, многие их свойства доступны через Инспектор объектов.

Доступ к значению поля

Объект поля, как и любой другой объект, имеет имя (название), определяемое его свойством `Name` типа `String`. Имя объекта `Field` зависит от того, является ли поле динамическим или статическим. По умолчанию для динамического поля имя объекта `Field` совпадает с именем соответствующего физического поля таблицы БД, для которого создан объект, и не может быть изменено. Имя статического поля является *составным* и по умолчанию образуется путем слияния имен набора данных и имени физического поля таблицы БД. Например, если для физического поля `Name` набора данных `Table1` с помощью Редактора полей создано статическое поле, то оно получит имя `Table1Name`. Программист может изменить это имя через Инспектор объектов, когда соответствующее статическое поле выбрано в Редакторе полей.

В отличие от имени объекта `Field`, свойство `FieldName` типа `String` содержит имя *физического поля*, заданное при создании таблицы. Не следует путать свойства `Name` и `FieldName`, они обозначают разные объекты и в общем случае могут не совпадать.

Пример обращения к полю:

```
Table1.FieldByName('Number').DisplayLabel := 'Количество';
Table1Number.DisplayLabel := 'Количество';
```

Здесь для статического поля `Number` возможны два способа обращения: по имени поля в наборе данных и по имени объекта `Field` поля.

Для определения порядкового номера поля в наборе данных можно использовать свойство `FieldNo` типа `Integer`, например, так:

```
var x: integer;
...
x := Table1.FieldByName('Date').FieldNo;
```

Для доступа к значению поля служат свойства Value и AsXXX. Свойство Value типа Variant представляет собой фактические данные в объекте типа TField. При выполнении приложения это свойство используется для чтения и записи значений в поле. Если программист обращается к свойству Value, то он должен самостоятельно обеспечивать преобразование и согласование типов значений полей и читаемых или записываемых значений. В табл. 17.3 приводятся возможные типы свойства Value для различных объектов типа TField.

Таблица 17.3. *Возможные типы свойства* `Value`

Тип объекта поля	Тип свойства Value
TField	Variant
TStringField, TBLOBField	String
TIntegerField, TSmallntField, TWordField, TAutoincField	Longint
TBCDField, TFloatField, TCurrencyField	Double
TBooleanField	Boolean
TDateTimeField, TDateField, TTimeField	TDateTime

Рассмотрим пример, в котором доступ к значению поля осуществляется с помощью свойства Value:

```
procedure TForm1.Button1Click(Sender: TObject);
var s: string;
    x: real;
begin
  // Доступ к полю по его имени в наборе данных
  s := Table1.FieldByName('Salary').Value;
  x := Table1.FieldByName('Salary').Value;
  Label1.Caption := s;
  Label2.Caption := FloatToStr(x);
  // Доступ к полю как к отдельному компоненту
  x := Table1Salary.Value;
  Label3.Caption := FloatToStr(x);
  // Поле вещественного типа,
  // в связи с чем следующая инструкция присваивания недопустима
  //  s := Table1Salary.Value;
end;
```

Здесь чтение значения поля Salary текущей записи набора данных Table1 выполняется несколькими способами. При доступе к полю по имени в наборе данных значение вещественного поля Salary можно читать и использовать и как строковое, и как вещественное значение. При доступе к полю как к отдельному компоненту тип переменной, в данном случае x, должен соответствовать типу поля.

Замечание

Доступ к полю по имени объекта типа TField, например, Table1Salary, возможен только для статических полей, которые существуют на этапе разработки приложения. Попытка использовать имя объекта динамического поля приводит к ошибке при компиляции, т. к. объект поля еще не создан.

Поскольку при доступе к полю с помощью свойства `Value` программист должен обеспечивать преобразование и согласование типов значений, то часто более удобно использовать варианты свойства `AsXXX`:

- `AsVariant` типа `Variant`;
- `AsString` типа `String`;
- `AsInteger` типа `Longint`;
- `AsFloat` типа `Double`;
- `AsCurrency` типа `Currency`;
- `AsBoolean` типа `Boolean`;
- `AsDateTime` типа `TdateTime`.

При использовании любого из этих свойств выполняется *автоматическое преобразование типа* значения поля к типу, соответствующему названию свойства. При этом преобразование должно быть допустимо, в противном случае возникает ошибка компиляции по несоответствию типов, например, при попытке прочитать логическое значение как целочисленное.

Приведем теперь пример, где доступ к значению поля происходит с помощью свойств `AsXXX`:

```
procedure TForm1.Button2Click(Sender: TObject);
var s: string;
    x: real;
begin
  // Доступ к полю по его имени в наборе данных
  s := Table1.FieldByName('Salary').AsString;
  x := Table1.FieldByName('Salary').AsFloat;
  Label1.Caption := s;
  Label2.Caption := FloatToStr(x);
  // Доступ к полю как к отдельному компоненту
  s := Table1Salary.AsString;
  x := Table1Salary.AsFloat;
  Label3.Caption := s;
  Label4.Caption := FloatToStr(x);
end;
```

Как и в предыдущем примере, чтение значения поля `Salary` осуществляется несколькими способами. Доступ к полю выполняется по имени поля и по имени объекта поля, а значение поля интерпретируется как строковое или как вещественное.

Замечание

Для того чтобы записать значение в поле, оно должно допускать модификацию, а набор данных должен находиться в соответствующем режиме, например, редактирования или вставки.

При необходимости программист может запретить модификацию поля, а также скрыть его, используя свойства `ReadOnly` и `Visible` типа `Boolean`. Сама возможность модификации данных в отдельном поле определяется значением свойства `CanModify` типа

Boolean. Напомним, что свойства ReadOnly и CanModify есть также у набора данных: они определяют возможность модификации набора данных (всех его полей) в целом.

Замечание

Даже если набор данных является модифицируемым и его свойство CanModify имеет значение True, для отдельных полей этого набора редактирование может быть запрещено, и любая попытка изменить значение такого поля вызовет исключение.

Если поле является невидимым (свойство Visible установлено в False), но разрешено для редактирования (свойство ReadOnly установлено в False), то можно изменить значения этого поля программно.

Рассмотрим управление видимостью поля и возможностью его модификации на примере:

```
Table1.FieldByName('Number').ReadOnly := True;
Table1.FieldByName('Salary').Visible := False;
```

Здесь для поля Number запрещаются любые изменения, а поле Salary скрывается, однако для него по-прежнему допускаются чтение и изменение значения.

Для полей, имеющих типы TBLOBField (BLOB-объект), TGraphicField (графическое изображение) и TMemoField (текст), доступ к их содержимому выполняется обычными для объектов данного типа способами. Например, для загрузки содержимого из файла можно использовать метод LoadFromFile.

Проверка типа и значения поля

При выполнении программы можно выяснить тип данных конкретного поля. Это удобно делать в случае, когда, например, типы данных полей таблиц БД заранее неизвестны.

Тип данных поля таблицы определяет свойство DataType типа TFieldType, принимающее следующие значения:

- ftUnknown (тип неизвестен или не определен);
- ftString (короткая строка длиной не более 255 символов);
- ftSmallInt (короткое целое число);
- ftInteger (целое число);
- ftWord (целое число без знака);
- ftBoolean (логическое значение);
- ftFloat (вещественное число);
- ftCurrency (денежное значение);
- ftBCD (число в формате BCD (двоично-десятичный формат));
- ftDate (дата);
- ftTime (время);
- ftDateTime (дата и время);
- ftBytes (байтовое значение фиксированной длины);

- ftVarBytes (байтовое значение переменной длины);
- ftAutoInc (автоинкрементное значение);
- ftBlob (BLOB-объект);
- ftMemo (текст Memo);
- ftGraphic (графический объект);
- ftFmtMemo (форматированный текст Memo);
- ftParadoxOle (поле OLE для таблицы Paradox);
- ftDBaseOle (поле OLE для таблицы dBase);
- ftTypedBinary (типизированное двоичное значение);
- ftCursor (курсор для хранимой процедуры Oracle);
- ftFixedChar (фиксированное количество символов);
- ftWideString (строка);
- ftLargeInt (длинное целое число);
- ftADT (значение абстрактного типа);
- ftArray (массив);
- ftReference (REF-поле);
- ftDataSet (DataSet-поле).

Программист с помощью специальных свойств может задать ограничения для вводимых в поля значений, а также проверить введенные значения.

Набор символов, допускаемых при вводе значений поля, зависит от типа данных поля:

- ftBoolean — все символы;
- ftSmallInt — цифры 0...9, знаки + и –;
- ftWord — цифры 0...9, знаки + и –;
- ftAutoInc — цифры 0...9, знаки + и –;
- ftDate — все символы;
- ftInteger — цифры 0...9, знаки + и –;
- ftTime — все символы;
- ftCurrency — цифры 0...9, знаки + и –, символы E или e, разделитель разрядов целой и дробной части;
- ftDateTime — все символы;
- ftFloat — цифры 0...9, знаки + и –, символы E или e, разделитель разрядов целой и дробной части;
- ftBCD — цифры 0...9, знаки + и –, символы E или e, разделитель разрядов целой и дробной части;
- ftString, ftVarBytes, ftBytes, ftBlob, ftDBaseOle, ftFmtMemo, ftGraphic, ftMemo, ftParadoxOle, ftTypedBinary, ftUnknown и ftCursor — все символы.

Отметим, что разделитель разрядов целой и дробной части числа (десятичный разделитель) зависит от установок Windows, выполненных через Панель управления.

Набор допустимых для поля символов содержит свойство `ValidChars` типа `TFieldChars`, который описан как множество символов:

```
type TFieldChars = set of Char;
```

Чтобы проверить, разрешен ли символ для ввода в качестве значения поля, удобно использовать метод `IsValidChar(InputChar: Char): Boolean`. Он возвращает значение `True`, если заданный параметром `InputChar` символ является для данного поля допустимым, и `False` — если символ не допускается.

Вот как осуществляется проверка допустимости символа в программе:

```
if not Table1.Fields[2].IsValidChar(Edit1.Text[1]) then
  MessageDlg('Символ "' + Edit1.Text[1] + '" не допустим!',
        mtError, [mbOK], 0);
```

Если первый символ текста, введенного в компонент `Edit1`, не является допустимым для поля (третьего в списке полей) набора данных `Table1`, то выдается сообщение об ошибке.

Замечание

Проверка на корректность символа не гарантирует отсутствие ошибок при вводе значений полей. Например, для числового поля типа `TIntegerField` не подойдет значение `12+3`, хотя каждый символ является допустимым.

Для полей числовых типов, например, объектов типа `TIntegerField` и `TFloatField` свойства `MinValue` и `MaxValue` позволяют установить минимальное и максимальное возможные значения, которые могут быть введены в соответствующее поле. Тип этих свойств определяется конкретным объектом типа `TField`.

Так устанавливается ограничение вводимых значений в программе:

```
Table1Price.MinValue  := 0;
Table1Price.MaxValue  := 9999.99;
Table1Number.MinValue := 0;
Table1Number.MaxValue := 1000;
```

Здесь для поля цены (вещественный тип) устанавливается допустимый диапазон `0...9999.99`, а для поля количества (целый тип) — диапазон `0...1000`.

Ограничения на вводимые значения можно определить также с помощью свойства `CustomConstraint` типа `String`. В этом свойстве допустимый диапазон задается с помощью конструкций, похожих на команды SQL или фильтры для отбора данных. Отличие заключается в том, что команды SQL и фильтры действуют при формировании наборов данных, а конструкция свойства `CustomConstraint` — при задании ограничений на вводимые значения. Например:

```
Table1Price.CustomConstraint := 'Price >= 0 and Price <= 9999.99';
Table1Number.CustomConstraint := 'Number >= 0 and Number <= 1000';
```

Здесь для полей цены и количества устанавливаются те же диапазоны допустимых значений, что и в предыдущем примере.

Если для поля некоторым способом задан диапазон допустимых значений, то при любой попытке установить поле в значение, выходящее за границы этого диапазона, вызывается исключение. При этом пользователю выдается соответствующее сообщение, а значение отвергается. Контроль значений относится не только к вводу со стороны пользователя, он также действует и при изменении значения поля программным способом.

С помощью свойства `ConstraintErrorMessage` типа `String` разработчик при желании может определить собственное сообщение, которое будет выдаваться при нарушении границ диапазона, например:

```
Table1Number.CustomConstraint := 'Number >= 0 and Number <= 10';
Table1Number.ConstraintErrorMessage :=
          'Количество должно находиться в диапазоне 0 .. 10';
```

Разработчик может не только ограничивать значения, вводимые в поля, но и задавать через свойство `DefaultExpression` типа `String` значение по умолчанию, которое автоматически заносится в поле при добавлении новой записи. Отметим, что это свойство строкового типа, поэтому его значения заключаются в кавычки.

Пример установки значения по умолчанию:

```
Table1Number.DefaultExpression := '100';
```

Чтобы проверить, действуют ли для поля ограничения, заданные в свойствах `CustomConstraint`, `DefaultExpression`, а также в `ImportedConstraint` (ограничения, налагаемые сервером), используется свойство `HasConstraints` типа `Boolean`. Это свойство, действующее во время выполнения приложения, принимает значение `True`, если установлено хотя бы одно из перечисленных ограничений.

Кроме проверки выхода за границы допустимых значений, программист может выполнить и более сложную проверку, используя обработчики событий `OnSetText` (типа `TFieldSetTextEvent`, `OnValidate`) и `OnChange` (типа `TFieldNotifyEvent`), возникающих в указанном здесь порядке при изменении значения поля. Типы этих событий описаны так:

```
type TFieldSetTextEvent = procedure (Sender: TField; const Text: string)
   of object;
type TFieldNotifyEvent = procedure (Sender: TField) of object;
```

Все упомянутые события возникают до изменения значения поля, поэтому программист может в необходимых случаях отказаться от внесения изменения. Отказ от изменения значения поля для этих событий выполняется различными способами. При использовании событий `OnValidate` и `OnChange` программист возбуждает исключение, в результате чего запись значения в поле не происходит. Если исключение не генерируется, то новое значение заносится в поле. Так как обработчик не получает нового значения поля, для доступа к нему нужно использовать свойства `Value` или `AsXXX`. Следует иметь в виду, что эти свойства возвращают новое (предлагаемое для утверждения) значение поля, которое может быть и не принято, в этом случае происходит возврат к старому значению.

При редактировании поля пользователем события `OnValidate` и `OnChange` не вызываются до тех пор, пока не будет выполнена попытка сохранить изменения, например, нажатием клавиши <Enter>.

Рассмотрим на примере, как осуществляется проверка вводимых значений.

В поле `Code` запрещается ввод значений `1` и `3`, которые зарезервированы для специальных целей. При попытке ввода неправильных значений после соответствующей проверки генерируется "тихое" исключение (посредством вызова процедуры `Abort`). В результате попытка ввода отвергается, при этом никакие сообщения пользователю не выдаются.

```
procedure TForm1.Table1NumberValidate(Sender: TField);
begin
  if (Table1.FieldByName('Code').AsString = '1') or
     (Table1.FieldByName('Code').AsString = '3')
    then Abort;
end;
```

Для выдачи сообщения о возникшем исключении в дополнение к процедуре `Abort` можно использовать, например, функцию `MessageDlg`, выводящую диалоговое окно. Кроме того, вместо вызова с помощью `Abort` "тихого" исключения можно вызвать обычное исключение, например, следующим образом:

```
Raise Exception.Create('Код не может равняться значениям 1 и 3!');
```

Для события `OnSetText` действует противоположный порядок принятия изменений и отказа от них. Параметр `Text` обработчика этого события содержит новое значение, которое предложено для занесения в поле, однако запись этого значения не осуществляется автоматически. Поэтому программист должен "вручную" занести предложенное значение в поле: если этого не сделать, то поле остается без изменений. Таким образом, по умолчанию в обработчике события `OnSetText` новое значение *отвергается*.

Вот еще один пример проверки вводимых значений:

```
procedure TForm1.Table1NumberSetText(Sender: TField; const Text: String);
begin
  if not ((Text = '1') or (Text = '3'))
    then Table1.FieldByName('Code').AsString := Text;
end;
```

Ряд полей, например, индексное поле, не могут быть пустыми, в этом случае у поля свойство `Required` типа `Boolean` имеет значение `True`. Для проверки полей, содержат ли они отличное от пустого значение, используется свойство `IsNull` типа `Boolean`. Это свойство возвращает значение `True`, если значение поля не задано, и значение `False` — в противном случае. Свойство `IsNull` можно использовать, например, в обработчиках событий `OnSetText`, `OnValidate` и `OnChange`:

```
procedure TForm1.Table1CodeValidate(Sender: TField);
begin
  if (Table1Code.Required) and (Table1Code.isNull)
     then Raise Exception.Create('Поле кода не может быть пустым!');
end;
```

Иногда перед занесением значения в некоторое поле необходимо проверить, является ли символ допустимым. Для этого можно использовать метод `IsValidChar`, проверяющий допустимость заданного символа, и свойство `ValidChars`, возвращающее набор допустимых для данного поля символов.

Замечание

Некоторые из описанных ограничений на значения полей также могут быть заданы при создании таблицы БД (запрет пустого значения и задание диапазона допустимых значений). В этом случае ограничения действуют на физическом уровне и не могут быть отменены.

Форматирование отображаемого значения поля

То, как выглядит значение поля в визуальных компонентах, зависит от типа поля, а также от системных установок Windows и BDE. Часто возникает необходимость изменить этот вид при отображении значений или их редактировании. Для этого используется *форматирование выводимой информации.*

Замечание

Названное форматирование относится только к отображению значений в визуальных компонентах и не затрагивает вид и размещение данных в таблице БД.

Свойство `DisplayFormat` типа `String` управляет отображением значений полей в визуальных компонентах, например, `DBGrid` и `DBEdit`. Это свойство действует для числовых полей, а также для полей даты и времени. В качестве значения свойства `DisplayFormat` задается маска, определяющая формат отображения поля.

Эта маска состоит из трех секций, разделенных символом `;`. Секции задают форму отображения положительных, отрицательных и нулевых значений соответственно. Если задана только одна секция, то она применяется для вывода всех значений. В маске можно использовать следующие управляющие символы:

- `0` — цифра числа, незначащие нули отображаются;
- `#` — цифра числа, незначащие нули гасятся;
- `.` — разделитель целой и дробной части числа;
- `,` — разделитель тысяч;
- `E+` или `e+` — разделитель мантиссы и порядка для чисел в форме с плавающей точкой, отображаются положительный и отрицательный знаки порядка;
- `E-` или `e-` — разделитель мантиссы и порядка для чисел в форме с плавающей точкой, отображается только отрицательный знак порядка;
- `"xx"` и `'xx'` — символы, выводимые без изменений; символы, заключенные в кавычки (двойные или апострофы), включаются в отображаемое значение;
- `;` — разделитель секций маски для положительных, отрицательных и нулевых значений.

Приведем примеры масок:

- `000E-00` или `000E+00` — для чисел в форме с плавающей точкой;
- `#####0.00` или `000000.00` — для чисел в форме с фиксированной точкой;
- `#####0.00' рублей'` — для денежных сумм;
- `####0;-####0;0` — отдельно для положительных, отрицательных и нулевых значений.

Свойство `EditFormat` типа `String` задает маску, используемую при форматировании значения числового поля перед его отображением для редактирования в визуальном компоненте. Формирование этой маски не отличается от формирования маски, задаваемой в свойстве `DisplayFormat`.

Свойство `EditMask` типа `String` задает маску, используемую при редактировании значения в визуальном компоненте. Маска позволяет ограничить число вводимых пользователем символов и тип вводимых символов (алфавитный, цифровой и т. д.). Кроме того, во вводимую информацию можно вставить дополнительные символы (разделители при вводе даты, времени и т. п.). С помощью свойства `EditMask` удобно вводить телефонные номера, даты, почтовые индексы и другую информацию подобного рода, имеющую заранее определенный формат.

Способы задания маски, содержащейся в свойстве `EditMask`, рассмотрены в *главе 3*, посвященной визуальным компонентам.

Приведенные свойства `DisplayFormat` и `EditMask` независимо друг от друга управляют форматированием значения поля при отображении и при редактировании. Кроме них, для объекта типа `TField` есть событие `OnGetText` типа `TFieldGetTextEvent`, в обработчике которого на программном уровне можно *одновременно* управлять форматом данных и при отображении, и при редактировании. Это событие генерируется при каждом обращении к свойствам `DisplayText` и `Text`. Тип события `OnGetText` описан так:

```
type TFieldGetTextEvent = procedure(Sender: TField; var Text: string;
   DisplayText: Boolean) of object;
```

Параметр `Text` содержит значение, которое выводится в визуальном компоненте. Программист должен задать это значение. Логический параметр `DisplayText` указывает, для каких целей выводится значение: для отображения (`DisplayText = True`) или для редактирования (`DisplayText = False`).

Рассмотрим пример, в котором используется форматирование отображаемых и редактируемых значений:

```
procedure TForm1.Table1NameGetText(Sender: TField; var Text: String;
                                   DisplayText: Boolean);
begin
  if DisplayText then Text := Table1Name.AsString
                 else Text := AnsiUpperCase(Table1Name.AsString);
end;

procedure TForm1.Table1PercentGetText(Sender: TField; var Text: String;
                                      DisplayText: Boolean);
begin
  if DisplayText then Text := Table1Percent.AsString + '%'
                 else Text := Table1Percent.AsString;
end;
```

Значение поля `Percent` отображается в визуальных компонентах со знаком `%`. При редактировании этого поля знак процента отсутствует. При редактировании значения поля `Name` оно выводится строчными буквами. Отметим, что если для перевода символов строки в верхний регистр вместо функции `AnsiUpperCase` использовать `UpperCase`, то символы русского алфавита будут преобразовываться некорректно.

Замечание

Если определен обработчик события `OnGetText`, то свойства `DisplayFormat` и `EditMask` не действуют.

Свойство `DisplayWidth` типа `Integer` определяет число символов, предназначенных для вывода значения поля в визуальном компоненте (обычно в сетке `DBGrid`), который связан с этим полем. По умолчанию значение этого свойства определяется шириной физического поля таблицы, заданной при его создании.

Замечание

Ширина столбца компонента-сетки `DBGrid` также зависит от значения свойства `Width` этого столбца, задающего ширину в пикселах.

Программист может изменить не только значение, но и заголовок поля, используемый как название столбца, например, в компоненте `DBGrid`. Заголовок поля определяется свойством `DisplayLabel` типа `String`. По умолчанию значением этого свойства является имя поля (свойство `FieldName`), однако, в отличие от последнего, свойство `DisplayLabel` доступно и для записи, и его можно использовать, например, для настройки пользователем интерфейса приложения.

Пример настройки заголовка поля:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1Name.DisplayLabel := Edit1.Text;
end;
```

Здесь при нажатии кнопки заголовку поля `Name` присваивается текст, введенный в компонент `Edit1`. Если это поле отображается в компоненте `DBGrid`, то соответствующий столбец сетки также изменяет свое значение на указанное.

Источник данных

Источник данных используется как промежуточное звено между набором данных и визуальными компонентами, с помощью которых пользователь управляет этим набором данных. В Delphi источник данных представлен компонентом `DataSource`.

Для указания набора данных, с которым связан источник данных, служит свойство `DataSet` типа `TDataSet` последнего. Визуальные компоненты связаны с источником данных через свои свойства `DataSource`. Обычно связь между источником и набором данных устанавливается на этапе проектирования в Инспекторе объектов, однако при необходимости эту связь можно установить или разорвать динамически. При смене у компонента `DataSource` набора данных визуальные компоненты автоматически подключаются к новому набору данных.

Указать набор данных можно, например, так:

```
DataSource2.DataSet := Nil;
DataSource1.DataSet := Table2;
```

Здесь для компонента `DataSource2` связь с набором данных разрывается, а для компонента `DataSource1` назначается набор данных `Table2`.

Для анализа состояния, в котором находится набор данных, можно использовать свойство `State` типа `TDataSetState`. При каждом изменении состояния набора данных для связанного с ним источника данных `DataSource` генерируется событие `OnStateChange` типа `TNotifyEvent`.

Если набор данных является редактируемым, то свойство `AutoEdit` типа `Boolean` определяет, может ли он автоматически переводиться в режим модификации при выполнении пользователем определенных действий. Например, для компонентов `DBGrid` и `DBEdit` таким действием является нажатие алфавитно-цифровой клавиши, когда компонент находится в фокусе ввода. По умолчанию свойство `AutoEdit` имеет значение `True`, и автоматический переход в режим модификации разрешен. Если необходимо защитить данные от случайного изменения, то одной из предпринимаемых мер является установка свойства `AutoEdit` в значение `False`.

Замечание

Значение свойства `AutoEdit` влияет на возможность редактирования набора данных только со стороны пользователя. Программно можно изменять записи независимо от значения этого свойства.

Без учета значения свойства `AutoEdit` пользователь может переводить набор данных в режим модификации путем нажатия кнопок компонента `DBNavigator`.

При изменении данных текущей записи генерируется событие `OnDataChange` типа `TDataChangeEvent`, описанного так:

```
type TDataChangeEvent = procedure (Sender: TObject; Field: TField) of object;
```

Параметр `Field` указывает на измененное поле; если данные изменены более чем в одном поле, то этот параметр имеет значение `Nil`. Следует иметь в виду, что в большинстве случаев событие `OnDataChange` генерируется и при переходе к другой записи. Это происходит, если хотя бы одно поле записи, ставшей текущей, содержит значение, отличное от значения этого же поля для записи, которая была текущей. Событие `OnDataChange` можно использовать, например, для контроля положения указателя текущей записи и выполнения действий, связанных с его перемещением. Это событие генерируется также при открытии набора данных.

Вот как осуществляется контроль перемещения указателя текущей записи:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  Label1.Caption := 'Запись номер ' + IntToStr(Table1.RecNo);
end;
```

При модификации текущей записи, кроме события `OnDataChange`, генерируется еще событие `OnUpdateData` типа `TNotifyEvent`. Оно возникает непосредственно перед записью данных в БД, поэтому в его обработчике можно предусмотреть дополнительный контроль и обработку введенных в поля значений, а также некоторые другие действия, например, отказ от изменения записи.

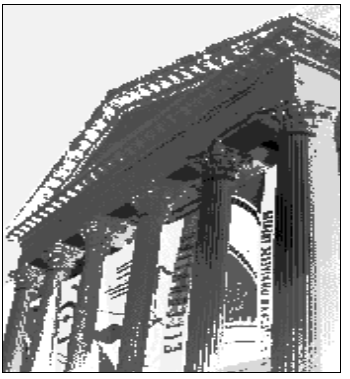
Иногда в визуальных компонентах требуется отключать отображение полей записей набора данных, например, при переборе записей в цикле их обработки, поскольку при этом возникает мелькание данных вследствие их быстрой смены. Для управления ото-

бражением записей можно использовать свойство `Enabled` типа `Boolean` источника данных.

Рассмотрим пример, демонстрирующий, как производится такое управление:

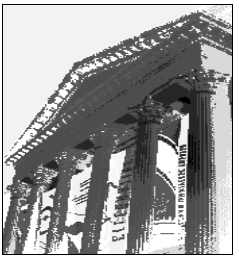
```
procedure TForm1.Button1Click(Sender: TObject);
var n: integer;
begin
  // Отключение отображения записей в визуальных компонентах
  DataSource1.Enabled := False;
  Table1.First;
  for n := 1 to Table1.RecordCount do begin
    // Обработка записи набора данных Table1
     ...
     Table1.Next;
  end;
  // Включение отображения записей в визуальных компонентах
  DataSource1.Enabled := True;
end;
```

В цикле перебираются все записи набора данных `Table1`. Перед началом цикла отображение записей в визуальных компонентах отключается, а после цикла — включается. Эти действия также можно выполнить, используя методы `DisableControls` и `EnableControls` набора данных.



ЧАСТЬ IV

Технологии доступа к данным



ГЛАВА 18

Визуальные компоненты для работы с данными

Визуальные компоненты для работы с данными расположены на странице **Data Controls** Палитры компонентов и предназначены для построения интерфейсной части приложения. Они используются для навигации по набору данных, а также для отображения и редактирования записей. Часто эти компоненты называют *элементами, чувствительными к данным*.

Одни визуальные компоненты для работы с данными предназначены для выполнения операций с полями отдельной записи, они отображают и позволяют редактировать значение поля текущей записи. К таким компонентам относятся, например, однострочный редактор `DBEdit` и графическое изображение `DBImage`.

Другие компоненты служат для отображения и редактирования сразу нескольких записей. Примерами таких компонентов являются сетки `DBGrid` и `DBCtrlGrid`, выводящие записи набора данных в табличном виде.

Визуальные компоненты для работы с данными похожи на соответствующие компоненты страниц **Standard**, **Additional** и **Win32** и отличаются, в основном, тем, что ориентированы на работу с БД и имеют дополнительные свойства `DataSource` и `DataField`. Первое из них указывает на источник данных — компонент `DataSource`, второе — на поле набора данных, с которым связан визуальный компонент. Например, однострочный редактор `DBEdit` работает так же, как однострочный редактор `Edit`, отображая строковое значение и позволяя пользователю изменять его. Отличие компонентов состоит в том, что в редакторе `DBEdit` отображается и изменяется значение определенного поля текущей записи набора данных.

Отметим, что для всех визуальных компонентов, предназначенных для отображения и редактирования значений полей, при изменении пользователем их содержимого набор данных автоматически переводится в режим редактирования. Произведенные с помощью этих компонентов изменения также автоматически сохраняются в связанных с ними полях при наступлении определенных событий, таких, например, как потеря фокуса визуальным компонентом или переход к другой записи набора данных.

При программном изменении содержимого этих визуальных компонентов набор данных не переводится в режим редактирования автоматически. Для этой цели в коде

должен предварительно вызываться метод `Edit` набора данных. Чтобы сохранить изменения в поле (полях) текущей записи, программист также должен предусмотреть соответствующие действия, например, вызов метода `Post` или переход к другой записи набора данных.

В табл. 18.1 приводятся так называемые стандартные, дополнительные и 32-разрядные визуальные компоненты, расположенные на страницах **Standard**, **Additional** и **Win32** Палитры компонентов, а также соответствующие им визуальные компоненты для работы с данными (страница **Data Controls**).

Таблица 18.1. *Соответствие визуальных компонентов, расположенных на разных страницах Палитры компонентов*

Компоненты страниц *Standard*, *Additional* и *Win32*	**Компоненты страницы *Data Controls***
`Label`	`DBText`
`Edit`	`DBEdit`
`Memo`	`DBMemo`
`RichEdit`	`DBRichEdit`
`ListBox`	`DBListBox`
`ComboBox`	`DBComboBox`
`CheckBox`	`DBCheckBox`
`RadioGroup`	`DBRadioGroup`
`Image`	`DBImage`
`StringGrid`	`DBGrid`
`Chart`	`DBChart`

Замечание

Часть компонентов (`QRLabel`, `QRRichEdit`, `QRDBRichEdit`, `QRImage`, `QRDBImage`, `QRShape`, `QRChart`), используемых для формирования отчетов в предыдущих версиях Delphi (изъятая страница **QReport** Палитры компонентов), тоже имеет свои аналоги среди других визуальных компонентов. Для обеспечения обратной совместимости компоненты страницы **QReport** размещены в пакете dclqrt70.bpl в папке \Delphi 7\Bin и при необходимости могут быть установлены в Палитру компонентов.

В этой главе мы рассмотрим особенности отдельных визуальных компонентов, предназначенных для работы с данными.

Отображение и редактирование значения логического поля

Логическое поле (поле логического типа) может содержать одно из двух значений: `True` (истина) или `False` (ложь). Разрешается использование прописных букв и сокращение вводимого значения, т. е. допустимы значения `True`, `true`, `tru`, `Tr`, `t` и т. д.

Для отображения и изменения значения логического поля можно использовать редактор DBEdit. Однако удобнее выполнять эти действия с помощью флажка (независимого переключателя) DBCheckBox, который позволяет "включить" или "выключить" значение логического поля.

Флажок DBCheckBox является аналогом рассмотренного ранее компонента CheckBox, поэтому здесь мы остановимся только на свойствах, характерных именно для этого флажка.

Компонент DBCheckBox выглядит на экране как квадратик (флажок) с текстовым заголовком (см. рис. 18.1). Если в нем находится галочка (при этом говорят, что флажок "включен" или "установлен"), то связанное с этим флажком логическое поле текущей записи содержит значение True. Если же квадратик пуст (флажок снят), то логическое поле текущей записи содержит значение False.

Флажок DBCheckBox можно применять также для отображения и редактирования строковых полей, если воспользоваться свойствами ValueChecked и ValueUnChecked.

Свойство ValueChecked типа String содержит строковые значения, которые устанавливают связанный с этим полем флажок во включенное состояние. Отдельные значения разделяются точкой с запятой. В качестве значений допускаются любые алфавитно-цифровые символы, в том числе русские буквы. Регистр алфавитных символов не различается, т. е. значения ДА и да считаются одинаковыми. Например:

```
DBCheckBox1.ValueChecked := 'True;T;Yes;Y;Да;Д';
```

Свойство ValueUnChecked типа String содержит строковые значения, которые устанавливают связанный с этим полем флажок в выключенное состояние. Значения задаются таким же образом, как и для свойства ValueChecked:

```
DBCheckBox1.ValueUnChecked := 'False;F;No;N;Нет;Н';
```

Если поле не содержит ни одного из значений, указанных в свойствах ValueChecked и ValueUnChecked, то флажок устанавливается в неопределенное состояние.

Отметим, что несмотря на наличие приведенных свойств, возможности флажка DBCheckBox по редактированию строковых полей намного меньше, чем у элемента DBEdit.

Отображение и выбор значения поля

В ряде случаев возникает необходимость ввода или отображения в поле одного из значений, входящих в состав фиксированного набора. С этой целью удобно использовать компонент DBRadioGroup (рис. 18.1), представляющий собой группу зависимых переключателей, из которых в каждый момент времени может быть включен (выбран) только один. Зависимые переключатели также называют просто переключателями; в форме они отображаются в виде кружка с текстовой надписью.

Отметим, что выбрать для поля одно из значений можно также с помощью списков, которые представляют для этих целей более широкие возможности.

Управление числом и названиями переключателей производится с помощью свойства Items типа TStrings, позволяющего получить доступ к отдельным переключателям в группе. Это свойство содержит строки, отображаемые как заголовки переключателей.

Отсчет строк в массиве начинается с нуля: `Items[0]`, `Items[1]` и т. д. Обычно задание значений этого свойства выполняется при разработке приложения с помощью Редактора строк. При выполнении приложения для манипуляции со строками (заголовками) можно использовать такие методы, как `Add` и `Delete`.

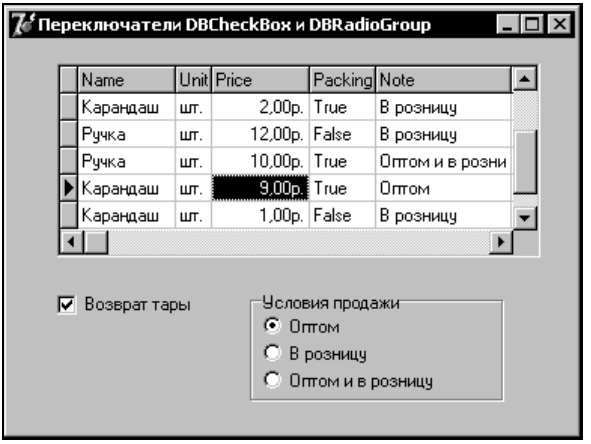


Рис. 18.1. Использование переключателей

Свойство `Values` типа `TStrings` содержит список значений поля, на которые должны реагировать переключатели группы. Управление этим списком осуществляется аналогично управлению списком `Items`. Если возможные значения свойства `Values` не заданы, то они выбираются из значений свойства `Items`, т. е. в этом случае значение, соответствующее переключателю, совпадает с названием этого переключателя.

Группа переключателей работает следующим образом. Переключатель включается при переходе к очередной записи, если значение связанного с ним поля содержит одно из значений, присутствующих в списке `Values`. Если же поле содержит значение, отсутствующее в списке возможных, то ни один переключатель не выбирается. Изменение значения поля происходит при выборе другого переключателя группы.

Замечание

Для поля, с которым связана группа переключателей, пользователь может выбрать значение только из списка. Попытки ввести в поле произвольное значение, например с помощью компонента `DBGrid`, блокируются.

В примере, приведенном на рис. 18.1, группа переключателей имеет заголовок **Условия продажи** и связана с полем `Note` набора данных. При перемещении на шестую запись в группе автоматически выбирается третий переключатель, имеющий название (и такое же значение) **Оптом и в розницу**. Если выбрать другой переключатель, например первый, то в поле `Note` шестой записи автоматически занесется новое значение — **Оптом**.

Замечание

Новое значение поля будет зафиксировано в наборе данных и отображено другими визуальными компонентами (`DBGrid` и подобными) после потери фокуса группой переключателей, например, в связи с переходом к другой записи.

Доступ к отдельному переключателю можно получить через свойство `ItemIndex` типа `Integer`, содержащее позицию (номер) переключателя, выбранного в группе в текущий момент. Это свойство используется для выбора отдельного переключателя или для определения, какой из переключателей является выбранным. Если свойство `ItemIndex` имеет значение `-1`, то не выбран ни один из переключателей.

Замечание

При программном выборе переключателя необходимо переводить набор данных в режим редактирования, а после установки нового значения поля закреплять сделанные изменения, например, вызовом метода `Post`.

Вот пример, иллюстрирующий выбор переключателя на программном уровне:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.Edit;
  DBRadioGroup1.ItemIndex := 1;
  Table1.Post;
end;
```

В приведенной процедуре при нажатии кнопки `Button1` в группе `DBRadioGroup1` выбирается второй переключатель. Предварительно набор данных переводится в режим редактирования, а после переключения и смены значения поля сделанные изменения сохраняются.

Отображение и выбор значения поля в списке

Чтобы отобразить поле текущей записи и выбрать этому полю новое значение, служат списки — компоненты `DBListBox`, `DBComboBox`, `DBLookupComboBox` и `DBLookupListBox`. По своей функциональности списки напоминают группу переключателей `DBRadioGroup`, однако предоставляют большие возможности. Список `DBComboBox` также обеспечивает возможность ввода в поле *произвольного значения*.

Простой и комбинированный списки

Компоненты `DBListBox` и `DBComboBox` позволяют пользователю выбирать один из строковых элементов. При выборе нужной строки простого списка `DBListBox` содержащееся в ней значение автоматически заносится в поле, с которым связан этот список. В дополнение к этому комбинированный список `DBComboBox` позволяет вводить произвольное значение.

Замечание

Новое значение не фиксируется в поле и, соответственно, не отображается в других визуальных компонентах, связанных с этим же набором данных. На рис. 18.2 в списке для поля `Unit` выбрано новое значение `Упаковка`, однако в сетке `DBGrid` это не отображается. Закрепление в поле нового значения, а также отображение этого значения в других визуальных компонентах, связанных с данным полем, произойдет при переходе к другой записи.

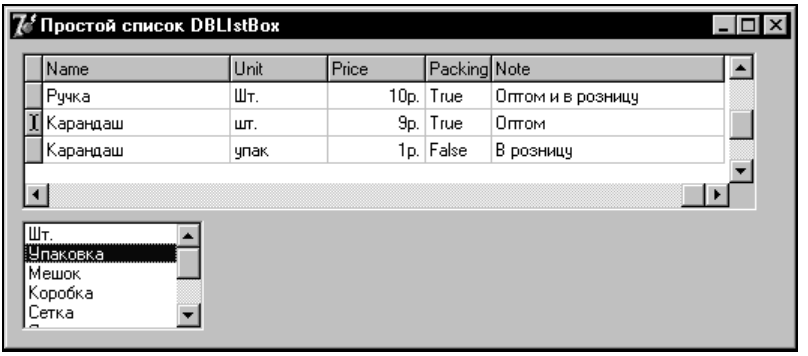


Рис. 18.2. Использование простого списка

Списки, сформированные по значениям поля набора данных

В простом и комбинированном списках перечень возможных значений поля формируется программистом и в общем случае не зависит от значений записей наборов данных. Иногда это может оказаться неудобным и привести к ошибкам ввода или выбора. В Delphi имеются компоненты `DBLookupListBox` и `DBLookupComboBox`, которые также предназначены для выбора значения в списке или непосредственного ввода значения в поле редактирования (только для `DBLookupComboBox`). Во многом эти компоненты похожи на простой список `DBListBox` и комбинированный список `DBComboBox` соответственно и отличаются от них только способом формирования списка возможных значений.

Принципы работы с компонентами `DBLookupComboBox` и `DBLookupListBox` практически такие же, как с полем выбора, которое было рассмотрено в *главе 17*. Основное отличие заключается в том, что поле выбора создается с помощью Редактора полей, и его основные свойства задаются в специальном окне, а в случае указанных компонентов для списка используется один из них (`DBLookupComboBox` или `DBLookupListBox`), а свойства устанавливаются, как правило, через Инспектор объектов.

Компонент `DBLookupComboBox` (или `DBLookupListBox`) связывается с полем "своего" набора данных через свойства `DataSource` и `DataField`, а список формируется с помощью свойств `ListSource` (типа `TDateSource`) и `DataField` (типа `String`), указывающих на второй набор данных и его поле, используемое для заполнения списка.

В свойстве `KeyField` типа `String` указывается поле второго набора данных, значение которого заносится в поле, связанное с компонентом списка.

Представление записей в табличном виде

Для вывода записей набора данных в табличном виде удобно использовать сетку, представленную в Delphi компонентом `DBGrid`. Внешний вид сетки соответствует внутренней структуре таблицы БД и набора данных, при этом строке сетки соответствует запись, а столбцу — поле.

С помощью сетки пользователь управляет набором данных, поля которого в ней отображаются. Для навигации по записям и их просмотра используются полосы прокрутки и клавиши перемещения курсора. Для перехода в режим редактирования поля достаточно установить на него курсор и нажать любую алфавитно-цифровую клавишу. Переход в режим вставки новой записи выполняется нажатием клавиши <Insert>, после чего можно заполнять поля. Вставка записи происходит в том месте, где находится указатель текущей записи. Изменения, сделанные при редактировании или добавлении записи, подтверждаются нажатием клавиши <Enter>, или переходом к другой записи, или отменяются нажатием клавиши <Esc>. Для удаления записи следует нажать комбинацию клавиш <Ctrl>+<Delete>.

Характеристики сетки

Несмотря на то что по своему виду сетка `DBGrid` похожа на сетку `StringGrid`, между ними есть значительные различия. Так, у сетки `StringGrid` можно устанавливать через соответствующие свойства число ее строк и столбцов. У сетки `DBGrid` числом строк управлять нельзя, т. к. она отображает все записи, имеющиеся в наборе данных.

Основным свойством сетки является свойство `Columns` типа `TDBGridColumns`, которое представляет собой массив (коллекцию) объектов `Column` типа `TColumn`, описывающих отдельные столбцы сетки.

Свойство `SelectedIndex` типа `Integer` задает номер текущего столбца в массиве `Columns`, а свойство `SelectedField` указывает на объект типа `TField`, которому соответствует текущий столбец сетки.

Свойство `FieldCount` типа `Integer`, доступное во время выполнения программы, содержит число видимых столбцов сетки, а свойство `Fields[Index: Integer]` типа `TField` позволяет получить доступ к отдельным столбцам. Индекс определяет номер столбца в массиве столбцов и принимает значения в интервале `0..FieldCount – 1`.

Свойства `Color` и `FixedColor` типа `TColor` задают цвета сетки и ее фиксированных элементов соответственно. По умолчанию свойство `Color` имеет значение `clWindow` (цвет фона Windows), а свойство `FixedColor` — значение `clBtnFace` (цвет кнопки).

Свойство `TitleFont` типа `TFont` определяет шрифт, используемый для вывода заголовков столбцов.

Доступ к параметрам сетки (например, для настройки) возможен через свойство `Options` типа `TGridOptions`. Это свойство представляет собой множество и принимает комбинации следующих значений:

- `dgEditing` (пользователю разрешается редактирование данных в ячейках);
- `dgAlwaysShowEditor` (сетка не блокирует режим редактирования);
- `dgTitles` (отображаются заголовки столбцов);
- `dgIndicator` (для текущей записи в начале строки выводится указатель);
- `dgColumnResize` (пользователь может с помощью мыши изменять размер столбцов и перемещать их);
- `dgColLins` (между столбцами выводятся разделительные вертикальные линии);

- dgRowLines (между строками выводятся разделительные горизонтальные линии);
- dgTabs (для перемещения по сетке можно использовать клавиши <Tab> и <Shift>+<Tab>);
- dgRowSelect (пользователь может выделить целую строку); при установке этого значения игнорируются значения dgEditing и dgAlwaysShowEditor;
- dgAlwaysShowSelection (ячейка остается выделенной, даже если сетка теряет фокус);
- dgConfirmDelete (при удалении строки выдается запрос на подтверждение операции);
- dgCancelOnExit (добавленные к сетке пустые строки (записи) при потере сеткой фокуса не сохраняются в наборе данных);
- dgMultiSelect (в сетке можно одновременно выделить несколько строк).

По умолчанию свойство Options содержит комбинацию значений [dgEditing, dgTitles, dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs, dgConfirmDelete, dgCancelOnExit].

При щелчке на ячейке с данными генерируется событие OnCellClick, а щелчок на заголовке столбца вызывает событие OnTitleClick. Оба события имеют тип TDBGridClickEvent, описываемый так:

```
type TDBGridClickEvent = procedure(Column: TColumn) of object;
```

Параметр Column представляет собой столбец, на котором был произведен щелчок.

При перемещении фокуса между столбцами сетки инициируются события OnColEnter и OnColExit типа TNotifyEvent, первое из которых возникает при получении столбцом фокуса, а второе — при его потере.

Если свойство Options содержит значение dgColumnResize, то пользователь может с помощью мыши перемещать столбцы сетки. При таком перемещении генерируется событие OnColumnMoved типа TMovedEvent, описываемого как:

```
type TMovedEvent = procedure(Sender: TObject; FromIndex,
                             ToIndex: Longint) of object;
```

Параметры FromIndex и ToIndex указывают индексы в массиве столбцов сетки, соответствующие предыдущему и новому положению перемещенного столбца соответственно.

Сетка DBGrid способна автоматически отображать в своих ячейках информацию, но при необходимости программист может выполнить и собственное отображение сетки. Это может понадобиться в случае, когда желательно выделить ячейку или столбец с помощью цвета или шрифта, а также вывести в ячейке, кроме текстовой, и графическую информацию, например, небольшой рисунок. Для программной реализации отображения сетки используется обработчик события OnDrawColumnCell типа TDrawColumnCellEvent, которое возникает при прорисовке любой ячейки. Тип события OnDrawColumnCell описан так:

```
type  TDrawColumnCellEvent = procedure(Sender: TObject;
   const Rect: TRect; DataCol: Integer; Column: TColumn;
   State: TGridDrawState) of object;
```

Параметр `Rect` содержит координаты ограничивающего ячейку прямоугольника, параметр `DataCol` определяет номер прорисовываемого столбца в массиве столбцов сетки, а параметр `Column` является объектом прорисовываемого столбца. Параметр `State` задает состояние ячейки и принимает комбинации следующих значений:

- `gdSelected` (ячейка находится в выбранном диапазоне);
- `gdFocused` (ячейка имеет фокус ввода);
- `gdFixed` (ячейка находится в фиксированном диапазоне).

Порядок вызова события `OnDrawColumnCell` зависит от значения свойства `DefaultDrawing` типа `Boolean`. Если свойство имеет значение `True` (по умолчанию), то перед генерацией события `OnDrawColumnCell` в ячейке отображается фон и выводится информация. Затем вокруг выбранной ячейки рисуется прямоугольник выбора. Если свойство `DefaultDrawing` имеет значение `False`, то сразу вызывается событие `OnDrawColumnCell`, в обработчике которого следует разместить операции по прорисовке области сетки.

Рассмотрим пример программной прорисовки сетки (листинг 18.1).

Листинг 18.1. Пример программной прорисовки сетки

```
// Свойство DefaultDrawing должно быть установлено в значение True
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
var r :TRect;
    s :string;
begin
  s := Table1.FieldByName(Column.FieldName).AsString;
  r := Rect;
  if (Column.FieldName = 'Debt') then begin
    DBGrid1.Canvas.Brush.Color := clRed;
    DBGrid1.Canvas.Font.Color := clYellow;
    DBGrid1.Canvas.Font.Style := [fsItalic];
    DBGrid1.Canvas.FillRect(Rect);
    if Table1.FieldByName(Column.FieldName).AsCurrency > 1000
    then begin
      ImageList1.Draw(DBGrid1.Canvas, Rect.Left + 2, Rect.Top + 2, 2);
      r.Left := r.Left + ImageList1.Width + 4;
    end;
    DBGrid1.Canvas.TextOut(r.Left, r.Top + 2, s);
  end
  else begin
    DBGrid1.Canvas.Brush.Color := clWhite;
    DBGrid1.Canvas.FillRect(Rect);
    DBGrid1.Canvas.Font.Color := clBlack;
    DBGrid1.Canvas.Font.Style := [];
    DBGrid1.Canvas.TextOut(r.Left, r.Top + 2, s);
  end;
end;
```

Здесь для столбца сетки, который соответствует полю Debt (Задолженность) набора данных, устанавливается красный цвет фона, а данные выводятся желтым цветом и курсивом. Кроме того, если задолженность превышает 1000 (рублей или других денежных единиц), то в поле Debt соответствующей записи слева от числа выводится рисунок, находящийся в компоненте ImageList1 (третий по счету). Список с рисунками можно подготовить заранее при разработке приложения и загрузить динамически во время выполнения программы.

При прорисовке ячеек используется свойство Canvas элемента DBGrid1, а также параметр Rect. Если обработчик события OnDrawColumnCell является общим для нескольких компонентов DBGrid, то при отображении их ячеек вместо названия конкретного компонента (в примере это DBGrid1) необходимо ставить конструкцию (Sender as TDBGrid) или TDBGrid(Sender). Например, инструкция

```
(Sender as TDBGrid).Canvas.Font.Color := clRed;
```

устанавливает красный цвет для символов ячеек сетки, указываемой параметром Sender.

Столбцы сетки

Отдельный столбец Column сетки представляет собой объект типа TColumn. По умолчанию для каждого поля набора данных, связанного с компонентом DBGrid, автоматически создается отдельный столбец, и все столбцы в сетке доступны. Такие столбцы являются *динамическими*. Для создания *статических* столбцов используется специальный Редактор столбцов. Если хотя бы один столбец сетки является статическим, то динамические столбцы уже не создаются ни для одного из оставшихся полей набора данных. Причем в наборе данных доступными являются статические столбцы, а остальные столбцы считаются отсутствующими. Изменить состав статических столбцов можно с помощью Редактора столбцов на этапе разработки приложения.

Взаимодействие между динамическими и статическими столбцами, а также Редактором столбцов аналогично взаимодействию между динамическими и статическими полями набора данных и Редактором полей.

Характеристики и поведение сетки и ее отдельных столбцов во многом определяются полями набора данных (а также соответствующими объектами типа TField), для которых создаются объекты типа TColumn.

Функционирование динамических столбцов зависит от свойств объекта поля: при изменении свойств объекта типа TField соответственно изменяются свойства объекта типа TColumn. К примеру, динамический столбец получает от поля такие характеристики, как имя и ширину.

Достоинством статических столбцов является то, что для их объектов можно установить значения свойств, отличные от свойств соответствующего поля и не зависящие от него. Например, если для некоторого статического столбца установить свое имя, то оно не будет меняться даже в случае, если с этим столбцом связывается другое поле набора данных. Кроме того, объекты типа TColumn статических столбцов создаются на этапе разработки приложения, и их свойства доступны через Инспектор объектов.

Для запуска Редактора столбцов (рис. 18.3) можно вызвать контекстное меню компонента `DBGrid` и выбрать в нем пункт **Columns Editor**. Редактор столбцов можно вызвать также щелчком мыши на свойстве `Columns` в окне Инспектора объектов.

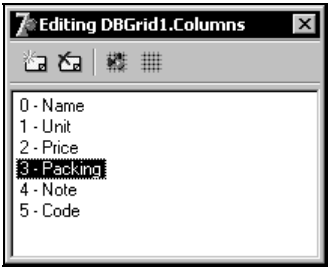


Рис. 18.3. Диалоговое окно Редактора столбцов

В заголовке Редактора столбцов выводится составное имя массива столбцов, например, `DBGrid1.Columns`. Под заголовком находится панель инструментов, видимостью которой можно управлять с помощью пункта **Toolbar** контекстного меню Редактора столбцов. Бóльшую часть Редактора столбцов занимает список статических столбцов, при этом столбцы перечисляются в порядке их создания (этот порядок может отличаться от исходного порядка полей в наборе данных).

Замечание

При изменении порядка столбцов сетки автоматически изменяется порядок связанных с ними полей набора данных, что необходимо учитывать при доступе к полям по номерам объектов типа `TField`, а не по именам объектов.

Первоначально список статических столбцов пуст, показывая тем самым, что все столбцы сетки являются динамическими. Редактор столбцов позволяет:

- создать статический столбец;
- удалить статический столбец;
- изменить порядок следования статических столбцов.

Кроме того, для любого выбранного в Редакторе статического столбца (объекта типа `TColumn`) через Инспектор объектов можно задать или изменить его свойства и определить обработчики его событий. Это допустимо потому, что соответствующие статическим столбцам объекты типа `TColumn` доступны уже на этапе разработки приложения.

Статический столбец можно создать следующими способами:

- нажатием кнопки  панели инструментов Редактора столбцов;
- выбором пункта **Add** контекстного меню Редактора столбцов;
- нажатием клавиши <Insert>.

В любом случае к списку добавляется строка, соответствующая новому статическому столбцу. В левой части строки содержится номер этого столбца в массиве столбцов, в правой — имя поля набора данных, с которым связан столбец. Сразу после добавления к списку столбец не связан ни с одним полем и вместо имени поля указывается

TColumn. При выполнении приложения подобный столбец окажется пустым. Чтобы связать столбец с каким-либо полем, необходимо установить значение его свойства FieldName.

Для добавления в список статических столбцов, соответствующих всем полям набора данных, следует нажать кнопку  панели инструментов Редактора столбцов или выбрать в его контекстном меню пункт **Add All Fields**.

Для удаления из списка статических столбцов необходимо их выделить, после чего выполнить одно из следующих действий:

- нажать кнопку  панели инструментов Редактора столбцов;
- выбрать пункт **Delete** контекстного меню Редактора столбцов;
- нажать клавишу <Delete>.

Вновь создаваемые статические столбцы получают значения свойств по умолчанию, зависящие также от полей набора данных, с которыми эти столбцы связаны.

Если значения свойств были изменены через Инспектор объектов и требуется восстановить их первоначальные значения, то это можно выполнить нажатием кнопки  или выбором пункта **Restore Defaults** (Восстановить параметры по умолчанию) контекстного меню Редактора столбцов.

Объект столбца доступен через свойство Columns типа TDBGridColumns. При проектировании приложения свойства этого объекта (т. е. столбца, выбранного в списке Редактора столбцов) доступны через Инспектор объектов.

Перечислим наиболее важные свойства объекта столбца.

- Alignment типа TAlignment — управляет выравниванием значений в ячейках столбца и может принимать следующие значения:
  - taLeftJustify (выравнивание по левой границе);
  - taCenter (выравнивание по центру);
  - taRightJustify (выравнивание по правой границе).
- Count типа Integer — указывает число столбцов сетки.
- Field типа TField — определяет объект поля набора данных, связанный со столбцом.
- FieldName типа String — указывает имя поля набора данных, с которым связан столбец. При установке этого свойства с помощью Инспектора объектов значение можно выбирать в списке.
- PickList типа TStrings — представляет собой список для выбора заносимых в поле значений. Текущая ячейка совместно со списком PickList образуют своего рода компонент ComboBox или DBComboBox. Если для столбца сформирован список выбора, то при попытке редактирования ячейки этого столбца справа появляется стрелка, при нажатии которой список раскрывается и позволяет выбрать одно из значений. При этом можно ввести в ячейку любое допустимое значение.

- `Title` типа `TColumnTitle` — представляет собой объект заголовка столбца. В свою очередь этот объект имеет такие свойства, как `Caption`, `Alignment`, `Color` и `Font`, определяющие название, выравнивание, цвет и шрифт заголовка соответственно.

Свойства столбца, управляющие форматированием, видимостью или возможностью модификации значений, не отличаются от соответствующих свойств поля.

Рассмотрим пример по установке свойств для столбцов сетки.

В наборе данных определено пять полей, для каждого из которых с помощью Редактора столбцов создан статический столбец. В листинге 18.2 приведен текст процедуры, выполняемой при создании формы приложения.

Листинг 18.2. Пример по установке свойств для столбцов сетки

```
// Значения этих свойств можно установить также через Инспектор объектов
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Скрытие первого столбца
  DBGrid1.Columns[0].Visible := False;
  // Установка параметров второго столбца
  DBGrid1.Columns[1].Title.Caption := 'Дата поступления';
  DBGrid1.Columns[1].Title.Alignment := taCenter;
  DBGrid1.Columns[1].Alignment := taCenter;
  // Скрытие третьего столбца
  DBGrid1.Columns[2].Visible := False;
  // Установка параметров четвертого столбца
  DBGrid1.Columns[3].Title.Caption := 'Количество';
  DBGrid1.Columns[3].Title.Alignment := taCenter;
  DBGrid1.Columns[3].Alignment := taRightJustify;
  // Установка параметров пятого столбца
  DBGrid1.Columns[4].Title.Caption := 'Примечание';
  DBGrid1.Columns[4].Title.Alignment := taCenter;
  DBGrid1.Columns[4].Title.Alignment := taLeftJustify;
  DBGrid1.Columns[4].PickList.Clear;
  DBGrid1.Columns[4].PickList.Add('Товар на складе');
  DBGrid1.Columns[4].PickList.Add('Некондиция');
  DBGrid1.Columns[4].PickList.Add('Срок реализации не лимитирован');
end;
```

Первый и третий столбцы устанавливаются невидимыми, для остальных столбцов задаются название заголовка и его выравнивание, а также выравнивание значений. Кроме того, для пятого столбца, соответствующего полю примечания, создан список выбора. Установка свойств столбцов произведена при создании формы, эти же действия могут быть проделаны и через Инспектор объектов. При выполнении приложения форма имеет вид, показанный на рис. 18.4.

В дальнейшем при использовании компонента `DBGrid` свойства его столбцов изменяться не будут, при этом состав и порядок следования столбцов сетки будут соответствовать составу и порядку следования полей набора данных, а в качестве заголовков столбцов сетки будут отображаться названия полей набора данных.

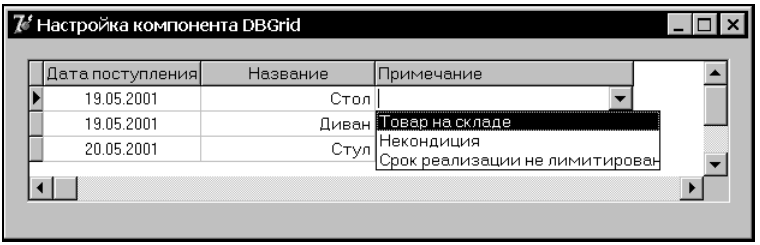


Рис. 18.4. Установка свойств для столбцов сетки

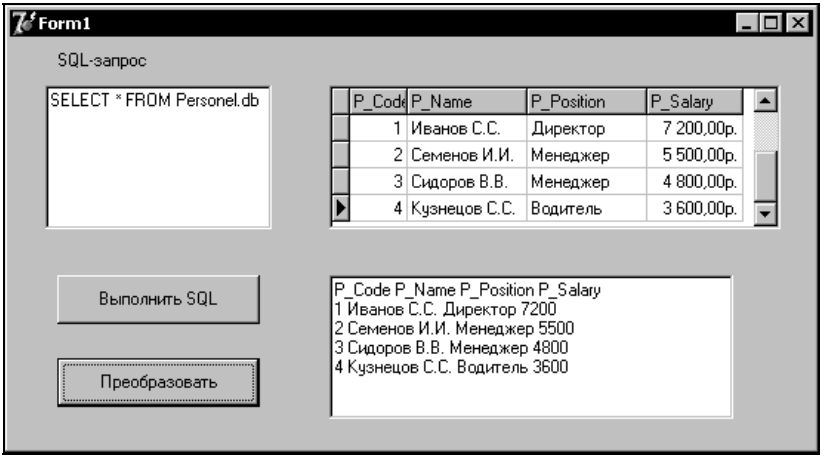


Рис. 18.5. Преобразование значений записей набора данных в текст

Рассмотрим еще один пример, в котором осуществляется преобразование значений записей набора данных в текст.

В качестве набора данных используется компонент `Query1`, SQL-запрос для которого вводится в многострочное поле редактирования `Memo1`. Выполнение запроса происходит при нажатии кнопки **Выполнить SQL** (`Button1`). Полученные в результате выполнения запроса записи отображаются в сетке `DBGrid1`. При нажатии кнопки **Преобразовать** (`Button2`) происходит последовательный просмотр полей всех записей набора данных (сетки) и преобразование их в текст, который помещается в многострочное поле редактирования `Memo2` (рис. 18.5).

В листинге 18.3 приведен код обработчиков событий нажатия кнопок.

Листинг 18.3. Пример преобразования значений записей набора данных в текст

```
// Выполнение SQL-запроса
procedure TForm1.Button1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.SQL.Assign(Memo1.Lines);
  Query1.Open;
end;
```

```
// Преобразование значений записей набора данных в текст
procedure TForm1.Button2Click(Sender: TObject);
var c, n : integer;
    s, rs: string;
begin
  Memo2.Clear;
  Query1.First;
  // Перебор всех записей набора данных
  for n := 1 to Query1.RecordCount do begin
    rs := ''; s := '';
    // Чтение названий столбцов сетки
    if n = 1 then begin
      for c := 0 to DBGrid1.Columns.Count — 1 do begin
        s := DBGrid1.Columns[c].FieldName + ' ';
        rs := rs + s;
      end;
      Memo2.Lines.Add(rs);
      rs := ''; s := '';
    end;
    // Чтение значений полей текущей записи
    for c := 0 to DBGrid1.Columns.Count — 1 do begin
      s := DBGrid1.Columns[c].Field.AsString + ' ';
      rs := rs + s;
    end;
    Memo2.Lines.Add(rs);
    Query1.Next;
  end;
end;
```

Для доступа к названиям и значениям полей набора данных использованы свойства `FieldName`, `Count` и `Field` столбцов сетки. При необходимости текст из поля редактирования `Memo2` можно скопировать в буфер.

Использование модифицированной сетки

Кроме компонента `DBGrid`, для управления записями таблицы предназначен компонент `DBCtrlGrid` — модифицированная сетка. Компонент `DBCtrlGrid` представляет собой несколько отдельных панелей, на которых располагаются визуальные компоненты (рис. 18.6).

Модифицированная сетка `DBCtrlGrid` может отображать несколько одинаковых панелей, но текущей является только одна из них. При проектировании приложения визуальные компоненты, например, `DBEdit` или `DBtext`, располагаются на одной (верхней) панели. Когда визуальные компоненты помещаются на панель модифицированной сетки, у них автоматически устанавливается нужное значение свойства `DataSource`, взятое из аналогичного свойства модифицированной сетки. При выполнении приложения компоненты, размещенные на одной панели, дублируются на другие панели сетки. На приведенной на рис. 18.7 сетке расположены четыре компонента `DBEdit` — для названия, единицы измерения, цены товара и примечания, компонент `DBtext` — для количества товара на складе и надпись `Label` с текстом `Наличие -`.

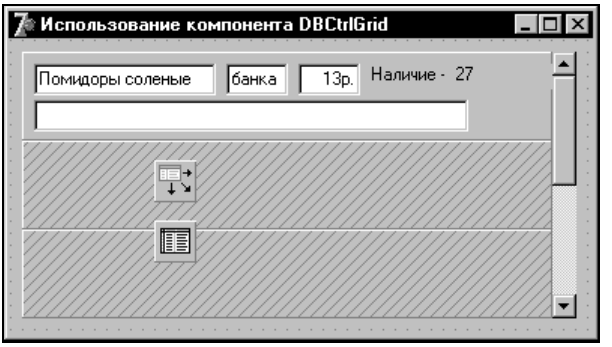


Рис. 18.6. Вид компонента `DBCtrlGrid` при проектировании приложения

Число панелей, одновременно видимых в модифицированной сетке, определяется свойством `PanelCount` типа `Integer`, доступным для чтения во время выполнения приложения. Свойство `PanelIndex` типа `Integer` указывает текущую панель, на которой находится просматриваемая запись набора данных. Установив это свойство в соответствующее значение, можно сделать текущей нужную панель.

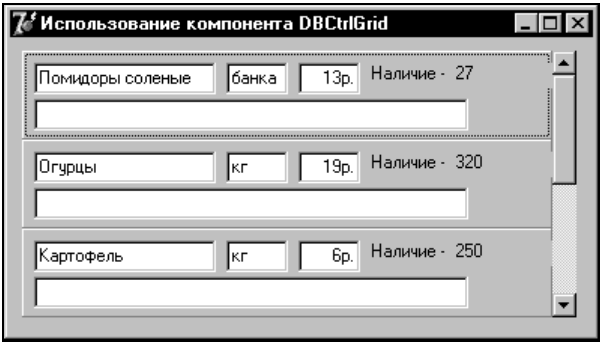


Рис. 18.7. Вид компонента `DBCtrlGrid` при выполнении приложения

Все панели сетки имеют одинаковые *размеры*, определяемые свойствами `PanelHeight` (высота) и `PanelWidth` (ширина) типа `Integer`. Каждая панель может иметь *рамку*, что определяется свойством `PanelBorder` типа `TDBCtrlGridBorder`, принимающим следующие значения:

- `gbNone` (рамки нет);
- `gbRaised` (трехмерная приподнятая рамка) — по умолчанию.

Число одновременно *видимых строк* и *столбцов* сетки задают свойства `RowCount` и `ColCount` типа `Integer`, значения которых по умолчанию равны трем и одному. Это соответствует трем панелям в одном столбце и наличию вертикальной полосы прокрутки. При задании более одного столбца у сетки появляется горизонтальная полоса прокрутки (рис. 18.8).

Свойство `Orientation` типа `TDBCtrlGridOrientation` при наличии нескольких столбцов определяет *порядок размещения записей* на панелях. Это свойство принимает следующие значения:

- `goVertical` (записи выводятся по горизонтали: слева направо и сверху вниз) — по умолчанию;
- `goHorizontal` (записи выводятся по вертикали: сверху вниз и слева направо).

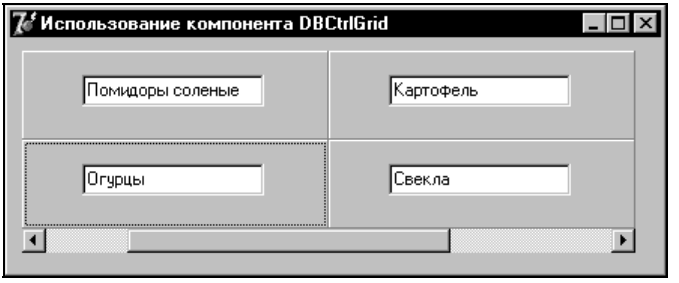


Рис. 18.8. Вид сетки с панелями в две строки и два столбца

На панели может отображаться *прямоугольник фокуса* (прямоугольная рамка), указывающий на текущую запись. Его отображением управляет свойство `ShowFocus` типа `Boolean`, которое по умолчанию имеет значение `True`, что соответствует отображению прямоугольника фокуса. Значение `False` свойства `ShowFocus` скрывает прямоугольник фокуса.

Свойства `AllowDelete` и `AllowInsert` типа `Boolean` управляют возможностями *удаления* текущей и *вставки* новой записи в набор данных, записи которого отображаются на панелях сетки. По умолчанию оба свойства имеют значение `True`, и пользователь имеет возможность удалять и вставлять записи.

Свойство `EditMode` типа `Boolean` определяет, находится ли набор данных в *режиме редактирования* (значение `True`). Пользователь управляет его значением с помощью действий с визуальными компонентами, разработчик может устанавливать его программно.

С помощью мыши и клавиатуры пользователь управляет записями набора данных посредством компонента `DBCtrlGrid` так же, как и в случае компонента `DBGrid`.

Программисту предоставляются дополнительные возможности управления набором данных, связанные с использованием метода `DoKey`. Процедура `DoKey(Key: TDBCtrlGridKey)` выполняет различные операции, которые задает параметр `Key`, принимающий следующие значения:

- `gkNull` (пустое действие);
- `gkEditMode` (переключение значения свойства `EditMode`);
- `gkPriorTab` (передача фокуса ввода следующему элементу формы);
- `gkNextTab` (передача фокуса ввода предыдущему элементу формы);
- `gkLeft` (переход на один столбец влево);
- `gkRight` (переход на один столбец вправо);
- `gkUp` (переход на одну панель вверх);
- `gkDown` (переход на одну панель вниз);

- gkScrollUp (переход на одну строку вверх);
- gkScrollDown (переход на одну строку вниз);
- gkPageUp (переход вперед на число записей, равное произведению числа строк и столбцов панелей сетки ColCount * RowCount);
- gkPageDown (переход назад на число записей, равное произведению числа строк и столбцов панелей сетки);
- gkHome (переход на первую запись);
- gkEnd (переход на последнюю запись);
- gkInsert (вставка новой записи методом Insert и переход в режим редактирования);
- gkAppend (вставка новой записи методом Append и переход в режим редактирования);
- gkDelete (удаление текущей записи);
- gkCancel (отмена режима редактирования).

При необходимости разработчик может выполнить *программную прорисовку* панелей, использовав для этого событие OnPaintPanel типа TPaintPanelEvent, возникающее непосредственно перед отображением панелей. Тип этого события описан как

```
type TPaintPanelEvent = procedure(DBCtrlGrid: TDBCtrlGrid;
                                  Index: Integer) of object;
```

Параметр Index указывает номер панели, отображение элементов которой выполняется в настоящий момент. Кроме прорисовки, в обработчике события OnPaintPanel можно выполнить другие действия, например, обработку связанных с панелью данных.

Использование навигационного интерфейса

Для управления набором данных можно использовать навигатор, который обеспечивает соответствующий интерфейс пользователя. По внешнему виду и организации работы навигатор похож на мультимедийный проигрыватель. В Delphi навигатор представлен компонентом DBNavigator (рис. 18.9).

Рис. 18.9. Навигатор

Навигатор содержит кнопки, обеспечивающие выполнение различных операций с набором данных путем автоматического вызова соответствующих методов. *Состав видимых кнопок* определяет свойство VisibleButtons типа TButtonSet, принимающее комбинации следующих значений (в скобках указан вызываемый метод):

- nbFirst — перейти к первой записи (First);
- nbPrior — перейти к предыдущей записи (Prior);
- nbNext — перейти к следующей записи (Next);
- nbLast — перейти к последней записи (Last);

- nbInsert — вставить новую запись (Insert);
- nbDelete — удалить текущую запись (Delete);
- nbEdit — редактировать текущую запись (Edit);
- nbPost — утвердить результат изменения записи (Post);
- nbCancel — отменить изменения в текущей записи (Cancel);
- nbRefresh — обновить информацию в наборе данных (Refresh).

По умолчанию в навигаторе видимы все кнопки.

Метод BtnClick(Index: TNavigateBtn) служит для имитации нажатия кнопки, заданной параметром Index. Тип TNavigateBtn этого параметра идентичен типу TButtonSet, возможные значения соответствующего параметра которого перечислены выше. В качестве примера приведем строку кода:

```
DBNavigator.BtnClick(nbNext);
```

В ней имитируется нажатие кнопки nbNext, вызывающей переход к следующей записи набора данных.

При нажатии кнопки nbDelete может появляться диалоговое окно, в котором пользователь должен подтвердить или отменить удаление текущей записи. Появлением *окна подтверждения* управляет свойство ConfirmDelete типа Boolean, по умолчанию имеющее значение True, т. е. окно подтверждения выводится. Если при отладке приложения установить это свойство в значение False, то запись будет удаляться без запроса подтверждения.

Свойство Flat типа Boolean управляет *внешним видом* кнопок. По умолчанию оно имеет значение False, и кнопки отображаются в объемном виде. При установке свойства Flat в значение True кнопки приобретают плоский вид, соответствующий современному стилю.

Подсказку для отдельной кнопки можно установить с помощью свойства Hints типа TString. По умолчанию список подсказок содержит текст на английском языке, который можно заменить на русский, вызвав Строковый редактор (String list editor). Подсказка для навигатора устанавливается через свойство Hint типа String. Напомним, что для отображения подсказок нужно присвоить значение True свойству ShowHint, по умолчанию имеющему значение False.

На практике часто вместо навигатора используются отдельные кнопки Button или BitBtn, при нажатии которых вызываются соответствующие методы управления набором данных. Например, в процедурах:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.Next;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Table1.Prior;
end;
```

при нажатии кнопок Button1 и Button2 выполняется переход к следующей и предыдущей записям набора данных Table1 соответственно.

Вывод графических изображений

Компонент DBImage (графическое изображение) предназначен для вывода изображений, содержащихся в графических полях БД. Если компонент DBImage связать с полем, не содержащим изображение, например с числовым, то в области компонента выводится имя этого поля. В случае, когда компонент DBImage не связан ни с одним полем, он отображает свое собственное имя (значение свойства Name).

Кроме вывода изображения, компонент также позволяет изменить (заменить) его, вставив нужный рисунок из буфера обмена Windows.

Основные свойства графического изображения рассмотрены в *главе 10*, посвященной графическим возможностям Delphi, здесь мы рассмотрим только дополнительные характеристики компонента DBImage.

Свойство AutoDisplay типа Boolean указывает, каким способом в компоненте DBImage отображаются *изменения* в связанном с ним поле. По умолчанию свойство имеет значение True, и содержимое графического поля отображается. Если свойство AutoDisplay установить в значение False, то при изменении значения поля, например, из-за перехода к другой записи, вместо изображения выводится имя поля. В этом случае для вывода графики нужно выполнить двойной щелчок на графическом компоненте или нажать клавишу <Enter>, когда на нем находится фокус ввода. Можно также вывести содержимое поля программно с помощью метода LoadPicture. Например:

```
DBImage1.AutoDisplay := False;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  DBImage1.LoadPicture;
end;
```

В данной процедуре при нажатии кнопки Button1 в графическом компоненте DBImage1 выводится содержимое графического поля текущей записи.

Компонент DBImage поддерживает работу с *буфером обмена* Windows, позволяя копировать изображение в буфер и вставлять изображение из буфера. Для выполнения этих действий используются обычные для Windows-программ комбинации клавиш: копирование в буфер — <Ctrl>+<Insert>, удаление в буфер — <Shift>+<Delete>, вставка из буфера — <Shift>+<Insert>.

Указанные действия также могут быть выполнены программно. Метод CopyToClipboard копирует изображение в буфер обмена, метод CutToClipboard вырезает (перемещает) изображение в буфер обмена, а метод PasteFromClipboard вставляет изображение из буфера обмена.

При использовании любого способа вставки из буфера новое изображение автоматически заменяет предыдущее содержимое компонента DBImage.

Рассмотрим в качестве примера программу для работы с фотоальбомом.

Пусть информация о фотографиях хранится в таблице Paradox, представляющей собой "электронный фотоальбом" и включающей следующие поля:

- код (автоинкрементное поле);
- название (символьное поле);
- дата (поле даты);
- изображение (графическое поле);
- описание (Memo-поле).

Для просмотра и редактирования альбома создано приложение, форма которого показана на рис. 18.10. Содержимое альбома выводится в соответствующих компонентах. В верхней части расположены поля редактирования `DBEdit`, одно из которых содержит название, а другое — дату фотографии. Снимок выводится в графическом компоненте `DBImage1`, а описание снимка — в многострочном редакторе `DBMemo1`. Для навигации по альбому используется навигатор `DBNavigator1`, расположенный в нижней части формы. Флажок `CheckBox1` управляет масштабированием изображения по размеру компонента `DBImage1`. По умолчанию этот флажок снят, и изображение не подстраивается под размеры графического компонента.

Рис. 18.10. Просмотр и редактирование электронного фотоальбома

Название, дата и описание фотографии редактируются обычным способом с помощью соответствующих компонентов `DBEdit` и `DBMemo`. Для редактирования снимка имеются кнопки, позволяющие вставлять изображение из файла и сохранять его в файле, а также обмениваться изображениями через буфер.

Нажатие кнопки **Открыть** вызывает появление диалога `OpenPictureDialog1` выбора файла для открытия. После выбора нужного файла содержащееся в нем изображение

загружается в компонент DBImage1. Фильтр диалога настроен на выбор графических файлов формата BMP, а также форматов ICO, EMF и WMF, которые при загрузке автоматически преобразуются в формат BMP.

Нажатие кнопки **Сохранить** открывает диалог SavePictureDialog1 выбора файла для сохранения. После выбора файла в него записывается изображение из компонента DBImage1. Фильтр диалога настроен на выбор графического файла формата BMP. Настройка фильтров обоих диалогов произведена при создании формы.

Кнопки **Копировать**, **Вырезать** и **Вставить** выполняют соответствующий обмен между графическим компонентом DBImage1 и буфером. Кнопка **Выход** прекращает работу приложения.

В листинге 18.4 приведен код модуля uAlbum формы Form1 приложения.

Листинг 18.4. Пример приложения для просмотра и редактирования альбома

```
unit uAlbum;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, DBCtrls, ExtCtrls, Db, DBTables, Grids,
  DBGrids, ExtDlgs, Mask;
type
TForm1 = class(TForm)
       DataSource1: TDataSource;
            Table1: TTable;
      DBNavigator1: TDBNavigator;
           Button1: TButton;
OpenPictureDialog1: TOpenPictureDialog;
SavePictureDialog1: TSavePictureDialog;
           Button2: TButton;
           Button3: TButton;
           Button4: TButton;
           Button5: TButton;
           Button6: TButton;
           DBMemo1: TDBMemo;
          DBImage1: TDBImage;
         CheckBox1: TCheckBox;
           DBEdit1: TDBEdit;
           DBEdit2: TDBEdit;

procedure Button1Click(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure Button3Click(Sender: TObject);
procedure Button4Click(Sender: TObject);
procedure Button5Click(Sender: TObject);
procedure Button6Click(Sender: TObject);
procedure FormCreate(Sender: TObject);
procedure CheckBox1Click(Sender: TObject);
```

```
private
{ Private declarations }
public
{ Public declarations }

end;
var Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  OpenPictureDialog1.Filter :=
   'Все файлы (*.bmp;*.ico;*.emf;*.wmf)|*.bmp;*.ico;*.emf;*.wmf';
              SavePictureDialog1.Filter := '*.bmp|*.bmp';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    DBImage1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  if SavePictureDialog1.Execute then
    DBImage1.Picture.LoadFromFile(SavePictureDialog1.FileName);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  DBImage1.CopyToClipboard;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  DBImage1.CutToClipboard;
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
  DBImage1.PasteFromClipboard;
end;

procedure TForm1.Button6Click(Sender: TObject);
begin
  Form1.Close;
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  DBImage1.Stretch := CheckBox1.Checked;
end;
end.
```

Отметим, что в отличие от компонента `Image`, у графического компонента `DBImage` нет свойства `Canvas`, позволяющего рисовать на поверхности изображения.

Построение диаграмм

Для построения диаграмм на основании информации, содержащейся в наборе данных, предназначен компонент-диаграмма `DBChart`. Он позволяет выводить диаграммы различных типов, в том числе объемные. Этот компонент является достаточно сложным и имеет большое количество разнообразных свойств, многие из которых являются объектами и также имеют свои свойства. На практике установка значений этих свойств выполняется при разработке приложения с помощью Редактора диаграмм, окно которого **Editing DBChart1** показано на рис. 18.11. Редактор позволяет оперировать со свойствами-объектами, информация о которых отображается на его страницах, и вызывается двойным щелчком на компоненте `DBChart` или через поле значения свойства-объекта в Инспекторе объектов (в этом случае активной становится страница, соответствующая выбранному свойству).

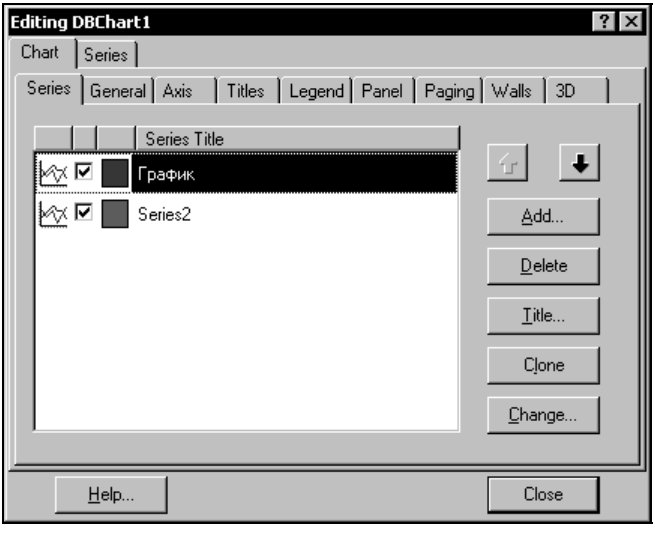


Рис. 18.11. Редактор диаграмм

Важнейшим свойством компонента `DBChart` является свойство `Series[Index:Longint]` типа `TChartSeries`, представляющее собой массив диаграмм, выводимых в области компонента. (Часто компонент `DBChart` содержит только одну диаграмму.)

Для каждой диаграммы можно установить:

- тип;
- название;
- оси;
- описание;
- источник данных
- и другие параметры.

Разработчик должен как минимум указать тип диаграммы и источник данных. Тип выбранной диаграммы (диаграмм) и ее название отображаются на странице **Chart | Series**

Редактора диаграмм (рис. 18.11). Для добавления новой диаграммы нужно нажать кнопку **Add**, в результате чего появляется окно, показанное на рис. 18.12. После выбора типа диаграммы, объемного или плоского варианта ее построения и нажатия кнопки **OK** диаграмма добавляется к значению свойства `Series` и отображается на соответствующей странице Редактора диаграмм.

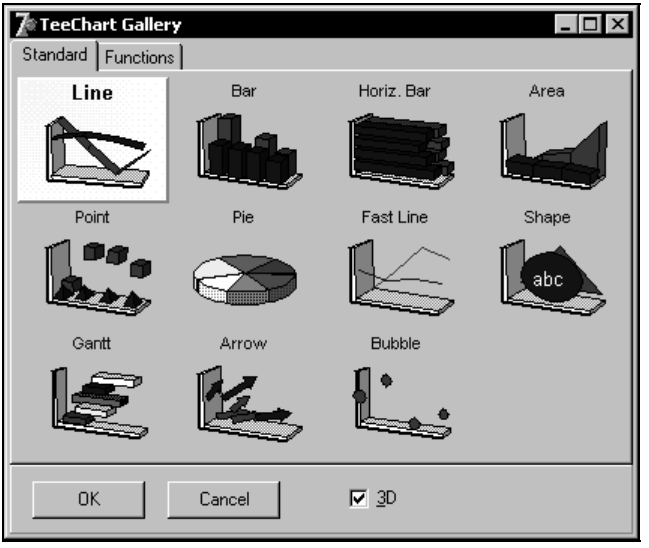


Рис. 18.12. Выбор типа диаграммы

Для выбранной диаграммы можно выполнить следующие действия:

- изменить имя по умолчанию (`Series1`, `Series2` и т. д.) — кнопка **Title**;
- изменить тип диаграммы — кнопка **Change**;
- скопировать диаграмму — кнопка **Clone**;
- удалить диаграмму — кнопка **Delete**.

Источник данных выбирается на странице **Series | DataSource** из следующих вариантов:

- **No Data** — значения, вводимые программно;
- **Random Values** — случайные числа;
- **Function** — значения, определяемые выбранной функцией;
- **DataSet** — значения набора данных; компонент `DBChart` отличается от компонента `Chart` именно тем, что для него в качестве источника данных можно использовать *набор данных*, т. е. `DBChart` является более универсальным компонентом.

Если выбран программный способ (вариант **No Data**) ввода значений, то при выполнении приложения нужно вызывать соответствующие методы. Для *управления значениями*, по которым строится диаграмма, часто используются методы `Add`, `Delete` или `Clear`.

Функция `Add(Const AValue: Double; Const ALabel: String; AColor: TColor): Longint` добавляет к диаграмме значение, указанное параметром `AValue`. Параметры `ALabel` и `AColor` содержат соответственно надпись значения и цвет, используемый при выводе. В качестве результата функция возвращает номер значения в массиве значений диаграммы. Кроме `Add`, есть несколько других методов, также позволяющих добавлять значения.

Процедура `Delete(ValueIndex : LongInt)` удаляет значение с номером, указанным параметром `ValueIndex`. Для удаления всех значений удобно использовать процедуру `Clear`.

В качестве примера рассмотрим, как осуществляется вывод графика.

Тип диаграммы (график) задан при разработке приложения. Вид графика при выполнении приложения показан на рис. 18.13. Используемые при его построении значения вводятся построчно в редакторе `Memo1`.

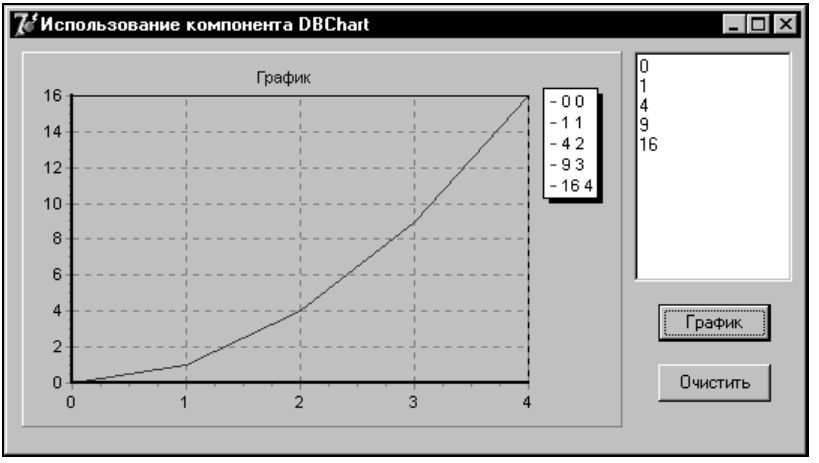


Рис. 18.13. Вывод графика

Далее приведены обработчики событий для кнопок формы приложения.

```
procedure TForm1.Button1Click(Sender: TObject);
var n: integer;
begin
  DBChart1.Title.Text.Add('График');
  for n := 0 to Memo1.Lines.Count – 1 do
    DBChart1.Series[0].Add(StrToFloat(Memo1.Lines[n]), IntToStr(n), clRed);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  DBChart1.Title.Text.Clear;
  DBChart1.Series[0].Clear;
end;
```

При нажатии кнопки **График** (`Button1`) значения из редактора `Memo1` заносятся в компонент `DBChart1`, отображающий график. Удаляется график нажатием кнопки **Очистить** (`Button2`).

Задание случайных чисел (вариант **Random Values**) в качестве источника данных для диаграммы бывает полезным при предварительной настройке диаграммы, например, при выборе ее типа или размера.

Использование функции (вариант **Function**) в качестве источника данных для диаграммы заключается в том, что диаграмма строится на основании обработки значений, взятых из двух и более других диаграмм (серий). В качестве функций обработки можно использовать **Copy**, **Average**, **Low**, **High**, **Divide**, **Multiply**, **Subtract** и **Add**. Например, если указать функцию **Subtract** (Вычитание) и две диаграммы-источника `Series1` и `Series2`, то каждое значение нашей диаграммы будет вычисляться как разность между соответствующими значениями диаграмм `Series1` и `Series2`.

При задании набора данных (вариант **Dataset**) в качестве источника данных для диаграммы (рис. 18.14) становится видимой панель для ввода информации о наборе данных.

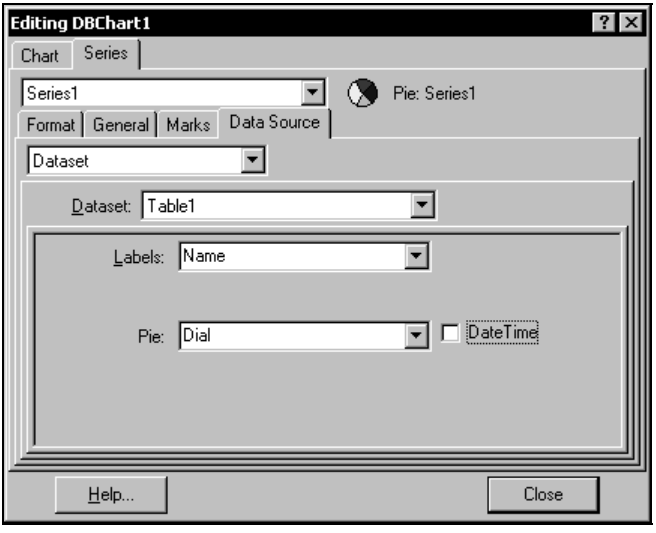


Рис. 18.14. Задание источника данных для диаграммы

В списке **Dataset** содержатся имена наборов данных, доступных в модуле той формы, в которой расположен компонент `DBChart`. В приведенном на рис. 18.14 примере — это набор данных `Table1`. В списке **Labels** выбирается имя поля (в примере `Name`), данные из которого используются в качестве надписей для обозначения секторов диаграммы, а в списке **Pie** — имя поля (в примере `Dial`), из которого выбираются данные для построения секторов диаграммы.

После закрытия окна Редактора диаграмма автоматически строится системой Delphi на основании записей, составляющих набор данных (рис. 18.15).

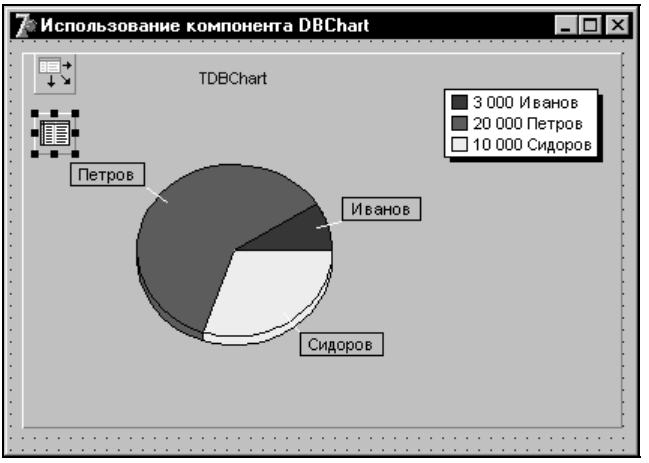
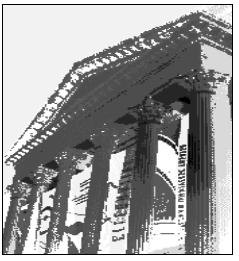


Рис. 18.15. Диаграмма, построенная на основании значений набора данных

При выполнении приложения диаграмма выглядит так же, как и при проектировании. При этом ее функционирование является динамическим, т. е. при изменении данных, содержащихся в наборе, диаграмма изменяется автоматически.

ГЛАВА 19



Навигационный доступ к данным с помощью механизма BDE

Напомним, что навигационный способ доступа заключается в обработке каждой отдельной записи набора данных. Достоинством этого способа является простота кодирования операций, а основной недостаток состоит в том, что приложение получает все записи набора независимо от того, сколько их требуется обработать на самом деле. Это приводит к большой нагрузке на сеть, особенно при интенсивном обмене данными. Поэтому применение навигационного способа доступа обычно ограничивается локальными БД.

В данной главе мы познакомимся с основными операциями, используемыми в локальных БД. Эти операции также могут быть применены и при организации работы с удаленными БД, если сеть имеет небольшое число пользователей. Операции с таблицами будут рассмотрены по отношению к наборам данных `Table` и `Query`, используемым при механизме BDE. Наряду с навигационным, язык структурированных запросов SQL и набор данных `Query` позволяют реализовать для локальных БД и реляционный доступ к данным, который подробно будет изложен в *главе 20*.

Если нет каких-либо специальных ограничений, то при работе с локальными БД более предпочтительным представляется набор данных `Table`, т. к. он работает несколько быстрее, чем `Query`. Для доступа к удаленным БД, наоборот, лучше использовать набор данных `Query`, т. к. с помощью SQL-запроса он позволяет реализовать реляционный доступ к данным, уменьшающий число передаваемых по сети записей.

При навигационном способе доступа операции выполняются с отдельными записями. Каждый набор данных имеет указатель текущей записи, т. е. записи, с полями которой могут быть выполнены такие операции, как редактирование или удаление. Компоненты `Table` и `Query` позволяют управлять положением этого указателя.

Навигационный способ доступа дает возможность осуществлять следующие операции:

- сортировку записей;
- навигацию по набору данных;
- редактирование записей;
- вставку и удаление записей;
- фильтрацию записей.

Отметим, что аналогичные операции применимы к набору данных и при реляционном способе доступа, реализуемом с помощью SQL-запроса.

Операции с таблицей БД

Кроме действий с отдельными записями, с помощью компонента `Table` можно выполнять также действия с таблицей БД в целом, например, создавать, удалять, переименовывать таблицы или устанавливать режимы доступа к ним.

Создание, удаление и переименование

Обычно таблицы создаются на этапе разработки приложения с помощью соответствующих инструментальных программ типа Database Desktop. Как правило, удаление таблицы также выполняется при разработке приложения, например, с помощью Проводника Windows. Использование инструментов позволяет достаточно удобно создавать таблицы, удалять их и изменять их структуру. Кроме того, программист может создать или удалить таблицу динамически, т. е. в процессе выполнения приложения. Такая потребность может возникнуть, например, при необходимости получить резервную или архивную копию всей таблицы или ее части.

Для *создания таблицы* используется метод `CreateTable`. В результате на диске появляется пустая таблица. Перед вызовом метода нужно подготовить необходимые данные, на основе которых создается таблица. Эти данные следует присвоить в качестве значений соответствующим свойствам набора данных. Перед вызовом метода `CreateTable` набор данных должен быть закрыт и установлены значения следующих свойств:

- `DatabaseName` (путь к файлам базы данных (каталог));
- `TableType` (тип таблицы);
- `FieldDefs` (описание полей);
- `IndexDefs` (описание индексов);
- `TableName` (название таблицы).

Свойство `TableName` задает имя физического файла таблицы, находящегося в каталоге, указанном для размещения БД (свойство `DatabaseName`).

Формат таблицы `TableName` типа `TTableName` может быть следующим:

- `ttDefault` (формат таблицы) — по умолчанию, определяется на основании расширений имен файлов таблиц:
  - db или отсутствует — таблица Paradox;
  - dbf — таблица dBase;
  - txt — таблица ASCII;
- `ttParadox` (таблица Paradox);
- `ttDBase` (таблица dBase);
- `ttFoxPro` (таблица FoxPro);
- `ttASCII` (таблица ASCII — текстовый файл, разбитый на столбцы).

Для новой таблицы в свойстве `FieldDefs` типа `TFieldDefs` обязательно должно быть определено хотя бы одно поле. Перед тем как приступить к описанию полей новой таблицы, значение этого свойства следует очистить, т. к. в нем может находиться информация о полях предыдущей таблицы, с которой был связан набор данных. Для *очистки значения* свойства `FieldDefs` можно применить метод `Clear`, а для *занесения информации о полях* новой таблицы — метод `Add`.

Процедура `Add(const Name: String; DataType: TFieldType; Size: Word; Required: Boolean)` добавляет к массиву полей описание нового *поля*. Параметр `Name` указывает название, а параметр `DataType` — тип поля, который можно выбрать в следующем списке: `ftUnknown`, `ftString`, `ftSmallint`, `ftInteger`, `ftWord`, `ftBoolean`, `ftFloat`, `ftCurrency`, `ftBCD`, `ftDate`, `ftTime`, `ftDateTime`, `ftBytes`, `ftVarBytes`, `ftAutoInc`, `ftBlob`, `ftMemo`, `ftGraphic`, `ftFmtMemo`, `ftParadoxOle`, `ftDBaseOle`, `ftTypedBinary`, `ftCursor`, `ftFixedChar`, `ftWideString`, `ftLargeint`, `ftADT`, `ftArray`, `ftReference`, `ftDataSet`. В перечисленных значениях префикс `ft` является сокращением от `Field Type` — тип поля, а последующая часть определяет собственно тип.

Параметр `Size` определяет размер поля; если для полей некоторых типов, например, поля даты (`ftDate`), размер не задается, то параметр `Size` принимает значение ноль. Логический параметр `Required` определяет, должно ли поле обязательно содержать значение (`True`) или может быть пустым (`False`).

Замечание

Если в наборе данных определены статические поля, то при вызове метода `CreateTable`, скорее всего, возникнет исключение. Это происходит из-за того, что в новой таблице задан новый состав полей и отсутствуют физические поля, с которыми были связаны созданные ранее статические поля.

В таблице можно определить *индексы*. Описание индексов заносится в свойство `IndexDefs` типа `TIndexDefs` (мы рассматривали его при описании набора данных `Table`) с помощью метода `Add`. Отметим, что после создания таблицы в ней можно удалить или создать индекс методами `AddIndex` и `DeleteIndex`.

Замечание

После задания новых индексов следует установить нужные значения для свойств `IndexName` и `IndexFieldName`, т. к. они могут указывать на индекс предыдущей таблицы, с которой был связан набор данных.

Для компонента `Query` действия, связанные с созданием таблицы, выполняются через запрос SQL.

Рассмотрим следующий пример (листинг 19.1).

Листинг 19.1. Пример создания таблицы через запрос SQL

```
procedure TForm1.btnNewTableClick(Sender: TObject);
begin
  // Закрытие набора данных
  Table1.Active := False;
```

```
  // Параметры таблицы БД
  Table1.DatabaseName := 'BDPlace';
  Table1.TableName := 'NewTable';
  Table1.TableType := ttParadox;
  // Описание полей таблицы
  Table1.FieldDefs.Clear;
  Table1.FieldDefs.Add('Code', ftAutoinc, 0, True);
  Table1.FieldDefs.Add('Name', ftString, 20, True);
  Table1.FieldDefs.Add('Date', ftDate, 0, False);
  // Описание индексов таблицы
  Table1.IndexDefs.Clear;
  Table1.IndexDefs.Add('', 'Code', [ixPrimary, ixUnique]);
  Table1.IndexDefs.Add('indName', 'Name', [ixCaseInsensitive]);
  // Создание таблицы
  Table1.CreateTable;
  // Установка текущего индекса
  Table1.IndexName := 'indName';
  // Открытие набора данных, связанного с новой (пустой) таблицей
  Table1.Active := True;
end;
```

Здесь в каталоге, указываемом псевдонимом `BDPlace`, создается новая таблица Paradox с именем `NewTable.db`. Для таблицы заданы три поля: код (номер), имя и дата. Поля кода и имени требуют обязательного заполнения при вводе или модификации записей созданной таблицы. По автоинкрементному полю номера построен главный ключ (без имени), а по полю имени — вторичный ключ, который после создания таблицы устанавливается как текущий.

Для *удаления* таблицы используется метод `DeleteTable`, в результате выполнения которого происходит физическое удаление всех файлов указанной таблицы. Путь и имя удаляемой таблицы определяют свойства `DatabaseName` и `TableName` набора данных. Перед удалением таблицы набор данных должен быть закрыт.

Для *переименования* таблиц dBase и Paradox можно использовать метод `RenameTable(const NewTableName: String)`, при выполнении которого переименовываются все файлы, относящиеся к таблице. Параметр `NewTableName` задает новое название таблицы. Напомним, что имя таблицы совпадает с именами файлов, а расширения имен файлов указывают на содержащуюся в них информацию, например, данные или индексы.

Установка уровня доступа

В случае многопользовательского доступа к БД для таблицы можно задать уровни доступа, которые будут действовать для других приложений. Уровень доступа определяет возможность записи данных в таблицу и их чтения.

Метод `LockTable(LockType: TLockType)` устанавливает *блокировку* для таблицы, при этом параметр `LockType` задает тип блокировки:

- `ltReadLock` (запрещены запись и чтение записей таблицы); этот режим является своего рода режимом монопольного доступа;

- `ltWriteLock` (запрещена запись в таблицу); другие приложения не могут выполнять модификацию записей таблицы, но чтение данных им разрешено.

Метод `UnLockTable(LockType: TLockType)` снимает установленную ранее блокировку.

Рассмотрим на конкретном примере, как осуществляется блокировка таблицы (листинг 19.2).

Листинг 19.2. Пример блокировки таблицы

```
procedure TForm1.Button1Click(Sender: TObject);
var n: longint;
begin
  try
    // Блокировка таблицы от внесения изменений
    Table1.LockTable(ltWriteLock);
    // Пересчет окладов
    Table1.First;
    for n := 1 to Table1.RecordCount do begin
       Table1.Edit;
       Table1.FieldByName('Salary').AsFloat :=
              Table1.FieldByName('Salary').AsFloat + 500;
       Table1.Next;
       Table1.Post;
    end;
  finaly
    // Отмена блокировки таблицы
    Table1.UnLockTable(ltWriteLock);
  end;
end;
```

Оклады всех сотрудников увеличиваются на `500` (рублей). На время пересчета таблица блокируется, что защищает ее от внесения изменений другими приложениями. При операциях с таблицей выполняется локальная обработка исключений, состоящая в отмене блокировки таблицы.

Сортировка набора данных

Порядок расположения записей в наборе данных может быть неопределенным. По умолчанию записи не отсортированы или сортируются, например, для таблиц Paradox по ключевым полям, а для таблиц dBase в порядке их поступления в файл таблицы.

С отсортированными записями набора данных работать более удобно. Сортировка заключается в упорядочивании записей по определенному полю в порядке возрастания или убывания содержащихся в нем значений. Сортировку можно выполнять и по нескольким полям. Например, при сортировке по двум полям сначала записи упорядочиваются по значениям первого поля, а затем группы записей с одинаковым значением первого поля сортируются по второму полю.

Сортировка наборов данных `Table` и `Query` выполняется различными способами. Здесь мы рассмотрим сортировку набора данных `Table`, для компонента `Query` вопросы сортировки рассмотрены в *главе 20*.

Сортировка наборов данных Table выполняется автоматически по текущему индексу. При смене индекса происходит автоматическое переупорядочивание записей. Таким образом, сортировка возможна по полям, для которых создан индекс. Для сортировки по нескольким полям нужно создать индекс, включающий эти поля.

Направление сортировки определяет параметр ixDescending текущего индекса, по умолчанию он выключен, и упорядочивание выполняется в порядке возрастания значений. Если признак ixDescending индекса включен, то сортировка выполняется в порядке убывания значений.

Напомним, что задать индекс (текущий индекс), по которому выполняется сортировка записей, можно с помощью свойств IndexName или IndexFieldNames. Эти свойства являются взаимоисключающими, и установка значения одного из них приводит к автоматической очистке значения другого. В качестве значения свойства IndexName указывается имя индекса, установленное при его создании. При использовании свойства IndexFieldNames указываются имена полей, образующих соответствующий индекс.

В связи с тем, что главный индекс (ключ) таблиц Paradox не имеет имени, выполнить сортировку по этому индексу можно только с помощью свойства IndexFieldNames.

Вот небольшой пример, иллюстрирующий сортировку с указанием имен индексов:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
     0: Table1.IndexName := 'indName';
     1: Table1.IndexName := 'indBirthDay';
  end;
end;
```

В качестве набора данных используется компонент Table1, а сортировка выполняется двумя способами: по индексу indName, созданному для поля Name, и по индексу indBirthDay, созданному для поля BirthDay.

Еще один пример сортировки, на этот раз с указанием имен индексных полей (для связанной с набором данных таблицы поле Code задано автоинкрементным и определено в качестве главного индекса):

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
     0: Table1.IndexFieldNames := 'Name';
     1: Table1.IndexFieldNames := 'Name;BirthDay';
     2: Table1.IndexFieldNames := 'Code';
  end;
end;
```

Здесь сортировка выполняется по следующим полям: Name (индекс indName), Name и BirthDay (индекс indNameBirthDay), Code (главный индекс).

А теперь рассмотрим более сложный пример сортировки. В качестве набора данных снова используется компонент Table1. Пользователь может управлять сортировкой его записей с помощью двух групп переключателей: в первой задается вид, а во второй —

направление сортировки. Сортировка выполняется после нажатия кнопки **Сортировать** (`btnSort`). Вид формы при проектировании показан на рис. 19.1.

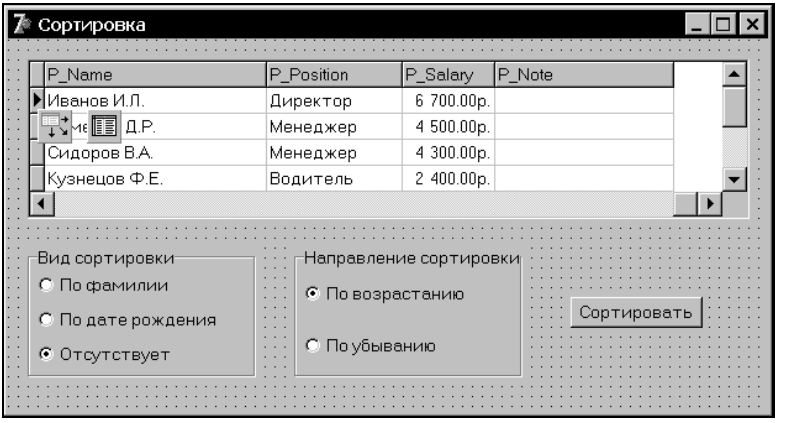


Рис. 19.1. Вид формы для сортировки набора данных

В связи с тем, что компоненты `Table` и `DataSource` являются невизуальными и при выполнении приложения не видны, их можно размещать в любом удобном месте формы, где они не мешают другим компонентам. Часто компоненты `Table` и `DataSource` помещаются на компоненте `DBGrid`, как это сделано в рассматриваемом примере.

В листинге 19.3 приводится обработчик события нажатия кнопки `btnSort`, вызывающей выполнение сортировки.

Листинг 19.3. Выполнение сортировки

```
procedure TForm1.btnSortClick(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0: Table1.IndexName := 'indName';
    1: Table1.IndexName := 'indBirthDay';
    2: Table1.IndexName := '';
  end;
  case RadioGroup2.ItemIndex of
    0: Table1.IndexDefs
      [Table1.IndexDefs.IndexOf(Table1.IndexName)].Options :=
         Table1.IndexDefs
        [Table1.IndexDefs.IndexOf(Table1.IndexName)].Options + [ixDescending];
    1: Table1.IndexDefs
      [Table1.IndexDefs.IndexOf(Table1.IndexName)].Options :=
         Table1.IndexDefs
           [Table1.IndexDefs.IndexOf(Table1.IndexName)].Options -
             [ixDescending];
  end;
end;
```

Поля, по которым сортируются записи, устанавливаются через свойство `IndexName`. При отсутствии сортировки этому свойству присваивается пустая строка. Для таблиц Paradox это означает сортировку по первому полю. Для таблиц dBase записи располагаются в порядке их поступления в файл таблицы.

Управление направлением сортировки осуществляется с помощью параметра `ixDescending` текущего индекса. Для определения номера текущего индекса в списке `IndexDefs` используется метод `IndexOf`.

Навигация по набору данных

Навигация по набору данных заключается в управлении указателем текущей записи (курсором). Этот указатель определяет запись, с которой будут выполняться такие операции, как редактирование или удаление.

Перемещение по записям

Перед перемещением указателя текущей записи набор данных автоматически переводится в режим просмотра. Если текущая запись находилась в режимах редактирования или вставки, то перед перемещением указателя сделанные в записи изменения вступают в силу, для чего набор данных автоматически вызывает метод `CheckBrowseMode`.

Для перемещения указателя текущей записи в наборе данных используются следующие методы:

- процедура `First` — установка на первую запись;
- процедура `Next` — установка на следующую запись (при вызове метода для последней записи указатель не перемещается);
- процедура `Last` — установка на последнюю запись;
- процедура `Prior` — установка на предыдущую запись (при вызове метода для первой записи указатель не перемещается);
- функция `MoveBy(Distance: Integer): Integer` — перемещение на число записей, определяемое параметром `Distance`. Если его значение больше нуля, то перемещение осуществляется вперед, если меньше нуля — то назад. При нулевом значении параметра указатель не перемещается. Если заданное параметром `Distance` число записей выходит за начало или конец набора данных, то указатель устанавливается на первую или на последнюю запись. В качестве результата возвращается число записей, на которое переместился указатель.

При перемещении указателя текущей записи учитываются ограничения и фильтр, определенные для набора данных. Таким образом, перемещение выполняется по записям набора данных, которые он содержит в текущий момент времени. Число записей определяется свойством `RecordCount`.

Значения указателя текущей записи изменяют также методы, связанные с поиском записей, например, метод `FindFirst`. Они рассматриваются ниже в этой главе.

Замечание

При изменении порядка сортировки набора данных расположение его записей может измениться, что чаще всего и происходит, но указатель по-прежнему указывает на первоначальную запись, даже если она находится на другом месте и имеет новое значение свойства RecNo.

Рассмотрим следующий пример:

```
procedure TForm1.Button3Click(Sender: TObject);
var s: real;
    n: integer;
begin
  s := 0;
  // Установка текущего указателя на первую запись
  Table1.First;
  for n := 1 to Table1.RecordCount do begin
    s := s + Table1.FieldByName('Salary').AsFloat;
    // Перемещение текущего указателя на следующую запись
    Table1.Next;
  end;
  Label2.Caption := FloatToStr(s);
end;
```

В приведенной процедуре перебираются все записи набора данных Table1, при этом в переменной s накапливается сумма значений, содержащихся в поле Salary. Перебор записей осуществляется с помощью метода Next, вызываемого в цикле. Предварительно с помощью метода First указатель устанавливается на первую запись. После выполнения кода указатель будет установлен на последнюю запись.

Отметим, что используемая для перебора записей переменная n имеет тип integer, совпадающий с типом longint свойства RecordCount.

Аналогичным образом можно перебрать все записи набора данных, начиная с последней. Естественно, при этом нужно вызывать методы Last и Prior.

Для *контроля положения* указателя текущей записи можно использовать свойство RecNo, которое содержит номер записи, считая от начала набора данных (для локальных таблиц dBase и Paradox).

Для таблиц Paradox свойство RecNo можно использовать еще и для перехода к записи с известным номером: такой переход выполняется установкой свойства RecNo в значение, равное номеру нужной записи. Например:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.RecNo := StrToInt(Edit1.Text);
end;
```

При нажатии кнопки Button1 указатель текущей записи набора данных Table1 устанавливается на запись, номер которой содержит редактор Edit1.

Для определения *начала* и *конца* набора данных при перемещении указателя текущей записи можно использовать свойства BOF и EOF типа Boolean соответственно. Эти свойства доступны для чтения при выполнении приложения. Свойство BOF показывает, на-

ходится ли указатель на первой записи набора данных. Этому свойству присваивается значение `True` при установке указателя на первой записи, например, сразу после вызова метода `First`. Свойство `EOF` показывает, находится ли указатель на последней записи набора данных. Этому свойству устанавливается значение `True` при размещении указателя на последней записи.

Замечание

Для пустого набора данных свойства `BOF` и `EOF` имеют значения `True`.

При изменении порядка сортировки или фильтрации, а также при удалении или добавлении записей значения свойств `BOF` и `EOF` могут изменяться. Например, если направление сортировки изменяется на противоположное, то первая запись становится последней.

При работе с таблицами одновременно нескольких приложений, когда постоянно добавляются или удаляются записи, значения свойств `BOF` и `EOF` соответствуют действительному состоянию набора данных в определенные моменты времени. Так, свойство `EOF` устанавливается в значение `True` сразу после выполнения метода `Last`. Если после этого другим приложением в конец набора данных добавлена новая запись, то значение свойства `EOF` становится неправильным.

Рассмотрим еще один пример, иллюстрирующий работу с указателем текущей записи:

```
procedure TForm1.Button2Click(Sender: TObject);
var s: real;
begin
  s := 0;
  Table1.First;
  while not Table1.EOF do begin
      s := s + Table1.FieldByName('Salary').AsFloat;
      Table1.Next;
  end;
  s := s + Table1.FieldByName('Salary').AsFloat;
  Label2.Caption := FloatToStr(s);
end;
```

Как и в предыдущем примере, здесь перебираются все записи набора данных `Table1` и подсчитывается сумма значений, содержащихся в поле `Salary`. Отличие заключается в том, что использован итерационный цикл с верхним окончанием, условием выхода из которого является достижение последней записи набора данных.

Иногда в цикле выполняются сложные действия. При этом на перебор записей набора данных может потребоваться достаточно большое время, в течение которого приложение (и компьютер) не реагирует на команды пользователя. Поэтому в циклы, надолго загружающие работой компьютер, рекомендуется вставлять вызов метода `Application.ProcessMessages`, обеспечивающий системе Windows возможность обрабатывать сообщения. Кроме того, программист должен предусмотреть досрочный выход из длительного цикла, например, с помощью переменной-признака.

Перед началом длительного цикла обработки записей БД и выполнения расчетов целесообразно выдать предупреждающее сообщение пользователю.

При перемещении по записям набора данных связанные с ним визуальные компоненты отображают изменения данных, причем смена отображения может происходить доста-

точно быстро, вызывая неприятное мелькание на экране. Чтобы избежать этого эффекта, можно программно до начала цикла перебора записей временно отключить набор данных от всех связанных с ним визуальных компонентов, а по окончании цикла снова подключить. Для этого предназначены методы `DisableControls` и `EnableControls`.

Рассмотрим следующий пример. Пусть в цикле перебираются все записи набора данных `Table1`, начиная с первой, и подсчитывается сумма значений, содержащихся в поле `Salary`. Для запуска и остановки процесса расчета используются кнопки **Начать** (`btnWorkStart`) и **Прервать** (`btnWorkBreak`). Поле индикатора `ProgressBar1` служит для показа хода работы, а процент перебранных записей отображается в надписи `lblWorkPercent`, находящейся рядом с индикатором. На рис. 19.2 показан вид формы во время выполнения приложения.

В листинге 19.4 содержится код модуля формы, приведенной на рис. 19.2.

Листинг 19.4. Пример перебора записей набора данных

```
unit uNavig;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Db, DBTables, Grids, DBGrids, StdCtrls, ExtCtrls,
  DBCtrls, ComCtrls;
type
  TForm1 = class(TForm)
          btnClose: TButton;
       DataSource1: TDataSource;
            Table1: TTable;
           DBGrid1: TDBGrid;
            Panel1: TPanel;
      ProgressBar1: TProgressBar;
      btnWorkStart: TButton;
      btnWorkBreak: TButton;
            Label1: TLabel;
    lblWorkPercent: TLabel;
            lblSum: TLabel;

    procedure btnCloseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnWorkStartClick(Sender: TObject);
    procedure btnWorkBreakClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }

  end;
var     Form1: TForm1;
    WorkBreak: boolean;
implementation
{$R *.DFM}
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  btnWorkStart.Enabled := True;
  btnWorkBreak.Enabled := False;
  WorkBreak := False;
end;

procedure TForm1.btnWorkStartClick(Sender: TObject);
label 10;
var s: real;
begin
  btnWorkStart.Enabled := False;
  btnWorkBreak.Enabled := True;
  s := 0;
  ProgressBar1.Max := Table1.RecordCount;
  Table1.DisableControls;
  Table1.First;
  while not Table1.EOF do begin
    s := s + Table1.FieldByName('Salary').AsFloat;
    // Без этого вызова не будет обрабатываться нажатие кнопки
    // btnWorkBreak
    Application.ProcessMessages;
    ProgressBar1.Position := Table1.RecNo;
    lblWorkPercent.Caption :=
      IntToStr(Round(ProgressBar1.Position/ProgressBar1.Max*100)) + '%';
    if WorkBreak then goto 10;
    Table1.Next;
  end;
  lblSum.Caption := 'Итого ' + FloatToStr(s);
10:
  Table1.EnableControls;
  ProgressBar1.Position := 0;
  lblWorkPercent.Caption := '';
  btnWorkStart.Enabled := True;
  btnWorkBreak.Enabled := False;
  WorkBreak := False;
end;

procedure TForm1.btnWorkBreakClick(Sender: TObject);
begin
  WorkBreak := True;
end;

procedure TForm1.btnCloseClick(Sender: TObject);
begin
  Close;
end;
end.
```

Цикл обработки запускается нажатием кнопки `btnWorkStart`. Перед началом цикла определяется максимальное значение индикатора `ProgressBar1`, переменная `WorkBreak`

устанавливается в `False`, а визуальные компоненты отключаются от набора данных `Table1`. В цикле происходит подсчет суммы, а также отображение хода выполнения работ в поле индикатора `ProgressBar1` и надписи `lblWorkPercent`. По окончании цикла результат отображается в надписи `lblSum`, и визуальные компоненты снова подключаются к набору данных `Table1`.

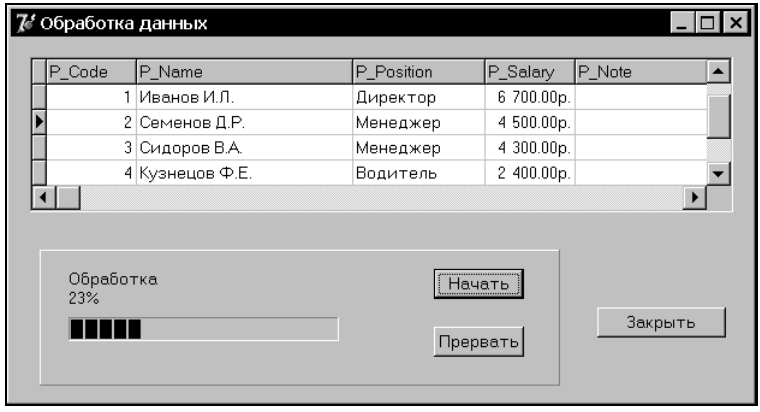


Рис. 19.2. Форма приложения, выполняющего перебор записей набора данных

Для прерывания обработки служит кнопка `btnWorkBreak`, при ее нажатии переменная-признак `WorkBreak` принимает значение `True`. Значение этой переменной проверяется в цикле обработки, и если оно равно `True`, то цикл прекращается путем вызова процедуры `Break`. Чтобы при выполнении цикла обрабатывались событие нажатия кнопки `btnWorkBreak` и другие события, в цикл включен вызов метода `ProcessMessages`.

При любом изменении положения указателя текущей записи для набора данных генерируются события `BeforeScroll` и `AfterScroll` типа `TDataSetNotifyEvent`.

Пользователь имеет возможность *перемещаться* по набору данных с помощью элементов управления, в качестве которых часто используются компоненты `DBGrid` и `DBNavigator`. Управление этими элементами с помощью мыши или клавиатуры приводит к автоматическому вызову соответствующих методов, перемещающих указатель текущей записи в заданное место. Например, после нажатия кнопок **First record**, **Prior record**, **Next record** или **Last record** компонента `DBNavigator1` косвенно вызываются методы `First`, `Prior`, `Next` или `Last`, перемещающие указатель текущей записи соответственно на первую, предыдущую, следующую или последнюю записи набора данных, с которым связан (через источник данных `DataSource`) компонент `DBNavigator1`.

Другим вариантом является размещение в форме элементов управления, например, кнопки `Button`, предназначенных для навигации по набору данных. Кроме того, часто в форме также размещаются элементы (обычно кнопки) для управления такими операциями, как редактирование, вставка записей, фильтрация и сортировка набора данных.

Рассмотрим в качестве примера форму приложения, в которой осуществляется перемещение по набору данных с помощью пяти кнопок (рис. 19.3). При нажатии какой-либо из этих кнопок вызывается соответствующий метод перемещения текущего указателя в заданном направлении. Например, при нажатии кнопки **Предыдущая запись**

(btnPrior) вызывается метод Prior. При перемещении указателя на произвольное число записей дополнительно используется обратный счетчик SpinEdit1, в поле которого вводится это число.

Рис. 19.3. Форма приложения, в которой осуществляется перемещение по набору данных

В листинге 19.5 приведен код модуля uNavig2 рассматриваемой формы приложения.

Листинг 19.5. Пример перемещения по набору данных

```
unit uNavig2;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Grids, DBGrids, Db, DBTables, Spin;
type
  TForm1 = class(TForm)
    DataSource1: TDataSource;
         Query1: TQuery;
        DBGrid1: TDBGrid;
       btnFirst: TButton;
        btnNext: TButton;
       btnPrior: TButton;
        btnLast: TButton;
      lblRecord: TLabel;
      btnMoveBy: TButton;
      SpinEdit1: TSpinEdit;

    procedure FormCreate(Sender: TObject);
    procedure btnFirstClick(Sender: TObject);
    procedure btnNextClick(Sender: TObject);
    procedure btnPriorClick(Sender: TObject);
    procedure btnLastClick(Sender: TObject);
    procedure Query1AfterScroll(DataSet: TDataSet);
    procedure btnMoveByClick(Sender: TObject);
```

```
  private
    { Private declarations }
  public
    { Public declarations }

  end;

var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  // Набор данных является редактируемым
  Query1.RequestLive := True;
  Query1.Close;
  Query1.SQL.Clear;
  Query1.SQL.Add('SELECT * FROM Goods.db');
  Query1.Open;
  SpinEdit1.Value := 0;
end;

// Переход на первую запись
procedure TForm1.btnFirstClick(Sender: TObject);
begin
  Query1.First;
end;

// Переход к следующей записи
procedure TForm1.btnNextClick(Sender: TObject);
begin
  Query1.Next;
end;

// Переход к предыдущей записи
procedure TForm1.btnPriorClick(Sender: TObject);
begin
  Query1.Prior;
end;

// Переход на последнюю запись
procedure TForm1.btnLastClick(Sender: TObject);
  begin
Query1.Last;
end;

// Перемещение на число записей, заданное в счетчике SpinEdit1
procedure TForm1.btnMoveByClick(Sender: TObject);
begin
  Query1.MoveBy(SpinEdit1.Value);
end;
```

```
// Вывод номера записи и пересчет границ счетчика
procedure TForm1.Query1AfterScroll(DataSet: TDataSet);
begin
  lblRecord.Caption := 'Запись номер ' + IntToStr(Query1.RecNo);
  SpinEdit1.MinValue := 1 — Query1.RecNo;
  SpinEdit1.MaxValue := Query1.RecordCount — Query1.RecNo;
end;
end.
```

Сразу после того, как положение указателя изменилось, в надписи `lblRecord` выводится номер текущей записи в наборе данных. Пересчитываются минимальное и максимальное значения обратного счетчика, определяющие диапазон допустимых значений, указываемых при вызове метода `MoveBy`. Эти действия кодируются в обработчике события `OnAfterScroll` для того, чтобы информация о текущей записи обновлялась сразу после перемещения текущего указателя. Событие `OnAfterScroll` происходит также при открытии набора данных, когда указатель текущей записи устанавливается на первую запись.

В качестве набора данных использован компонент `Query1`, задание основных свойств которого выполняется при создании формы. SQL-запрос (`SELECT * FROM Goods.db`) указан в свойстве `SQL`. Он обеспечивает получение всех полей и записей таблицы `Goods.db`. Свойство `RequestLive` установлено в значение `True`, чтобы обеспечить возможность редактирования набора данных. Таким образом, в рассмотренном примере компонент `Query` практически не отличается от компонента `Table` в плане работы с редактируемым набором данных, связанным с одной таблицей.

Иногда при подключении к одной таблице нескольких наборов данных возникает необходимость *синхронизации* их текущих указателей. Для этих целей используется метод `GotoCurrent(Table: TTable)`, устанавливающий указатель на текущую запись в наборе данных `Table`. Например:

```
Table1.GotoCurrent(Table2);
```

Здесь в наборе данных `Table1` текущий указатель устанавливается на запись, которая является текущей в наборе данных `Table2`.

Переход по закладкам

Побочным эффектом выполнения ряда операций с наборами данных является изменение положения указателя текущей записи. Часто этот эффект нежелателен, т. к. после выполнения такой операции указатель находится не в том месте, где был до начала операции. При этом приходится снова отыскивать нужную запись и позиционировать на ней указатель, что неудобно даже при небольшом количестве записей.

Примером такой операции является рассмотренный ранее расчет суммы значений по полю `Salary`, при котором выполняется перебор всех записей в прямом или обратном порядке. По окончании цикла указатель находится на первой или последней записи, а не там, где он был до начала суммирования.

Замечание

Положение указателя текущей записи может измениться также при формировании отчета из записей набора данных.

Для восстановления прежнего положения текущего указателя можно использовать, например, поиск записей: до начала операции данные текущей записи запоминаются, а после ее выполнения осуществляется поиск записи по этим данным. Похожим вариантом восстановления прежнего положения указателя является использование номера записи в наборе данных, который доступен через свойство `RecNo`.

Переход на определенную запись может понадобиться также для позиционирования указателя на запись, информация о которой была сохранена. Осуществляемые при этом действия не отличаются от указанных выше. Информация, необходимая для выполнения последующего перехода к требуемой записи, может запоминаться предварительно в удобный момент, например, при просмотре или редактировании записи.

Вот пример, иллюстрирующий, как восстановить положение текущего указателя:

```
var RecordNumber: integer;
...
// Запоминание номера текущей записи
RecordNumber := Query1.RecNo;
// Операция, изменяющая положение указателя
// Переход к записи с запомненным номером
Query1.First;
Query1.MoveBy(RecordNumber — 1);
```

Здесь для перемещения текущего указателя используется номер записи, запоминаемый в переменной `RecordNumber`.

В таблицах Paradox для возврата к записи можно установить свойство `RecNo` набора данных в предварительно сохраненное значение. Например:

```
var RecordNumber: integer;
...
// Запоминание номера текущей записи
RecordNumber := Table1.RecNo;
// Операция, изменяющая положение указателя
// Переход к записи с запомненным номером
Table1.RecNo := RecordNumber;
```

Вопросы, связанные с поиском записей, рассматриваются в данной главе позднее.

Кроме описанных способов, для перехода на определенную запись можно использовать *закладки* — специальные пометки каких-либо записей. Закладка имеет тип `TBookmark`, представляющий собой обычный указатель `Pointer`, с помощью которого можно ссылаться на различные объекты, в данном случае — на записи набора данных.

Перед использованием закладку нужно *создать*, т. е. определить запись, с которой эта закладка связана. Для этого предназначена функция `GetBookmark: TBookmark`, которая создает закладку на текущей записи набора данных. Возвращаемую закладку обычно запоминают в переменной типа `TBookmark` и впоследствии используют для перехода к помеченной ею записи.

Иногда закладка может неверно указывать позицию связанной с ней записи, при этом говорят о *нестабильности* закладки. Стабильность закладки в значительной степени зависит от форматов таблиц. Так, для таблиц dBase закладка всегда стабильна; для таблиц Paradox закладка стабильна, если у таблицы определен главный ключ. Таким образом, закладка устойчиво указывает на связанную с ней запись в тех случаях, когда существует признак, позволяющий однозначно различать записи. В противном случае закладка может быть нестабильной и ошибочно указывать на другую запись набора данных. Для проверки стабильности закладки используется функция `DBIGetCursorProps` (модуль `DBE`).

На установленную закладку можно перейти с помощью метода `GotoBookmark`. Процедура `GotoBookmark(Bookmark: TBookmark)` позиционирует указатель текущей записи на закладку, указанную параметром `Bookmark`. Если закладка предварительно не создана и ее значение равно `Nil`, то текущий указатель не перемещается.

Для удаления закладки используется метод `FreeBookmark`. Процедура `FreeBookmark(Bookmark: TBookmark)` сбрасывает значение закладки в `Nil` и освобождает занимаемую ею память.

Замечание

Попытка перейти на удаленную закладку путем вызова метода `GotoBookmark` приводит к возникновению исключения. К такому же эффекту приводит попытка повторного освобождения закладки методом `FreeBookmark`.

Целесообразно перед использованием закладки уточнить, существует ли она. Функция `BookmarkValid(Bookmark: TBookmark): Boolean` проверяет наличие закладки, заданной параметром `Bookmark`. Если закладка найдена, то в качестве результата функция возвращает значение `True`, в противном случае — `False`.

Можно устанавливать и использовать несколько закладок.

Вот пример, иллюстрирующий использование закладок для перехода к нужным записям:

```
procedure TForm1.Button1Click(Sender: TObject);
// Описание переменной типа закладка
var bm: TBookmark;
begin
  // Создание закладки bm
  bm := Table1.GetBookmark;
  Table1.DisableControls;
  Table1.First;
  while not Table1.EOF do begin
      // Обработка записей
      Table1.Next;
  end;
  // Переход на закладку bm
  if Table1.BookmarkValid(bm) then Table1.GotoBookmark(bm);
  // Освобождение закладки bm
  if Table1.BookmarkValid(bm) then Table1.FreeBookmark(bm);
  Table1.EnableControls;
end;
```

Перед началом цикла перебора записей на текущей записи набора данных `Table1` устанавливается закладка `bm`. По окончании цикла для восстановления прежнего положения текущего указателя выполняется переход на закладку, и закладка удаляется. Перед переходом на закладку проверяется ее существование.

Аналогичным образом закладки можно использовать и для перехода к заранее выбранным и отмеченным записям.

Фильтрация записей

Фильтрация — это задание ограничений для записей, отбираемых в набор данных. Напомним, что набор данных представляет собой записи, выбранные из одной или нескольких таблиц. Состав записей в наборе данных в данный момент времени зависит от установленных ограничений, в том числе от фильтров. Система Delphi дает возможность осуществлять фильтрации записей:

- по выражению;
- по диапазону.

По умолчанию фильтрация записей не ведется, и набор данных `Table` содержит все записи связанной с ним таблицы БД, а в набор данных `Query` включаются все записи, удовлетворяющие SQL-запросу, содержащемуся в свойстве `SQL`.

Замечание

При включении фильтрация записей действует в дополнение к другим ограничениям, например, SQL-запросу компонента `Query` или ограничению, налагаемому отношением "главный — подчиненный" между таблицами БД. Отметим, что для компонента `Query` SQL-запрос является средством отбора записей в набор данных, а фильтрация дополнительно ограничивает состав этих записей.

Фильтрация по выражению

При использовании фильтрации по выражению набор данных ограничивается записями, удовлетворяющими выражению фильтра, задающему условия отбора записей.

Достоинством фильтрации по выражению является то, что она применима к любым полям, в том числе и к неиндексированным. В связи с тем, что в процессе отбора просматриваются все записи таблицы, фильтрация по выражению эффективна при небольшом количестве записей.

Для задания выражения фильтра используется свойство `Filter` типа `String`. Фильтр представляет собой конструкцию, в состав которой могут входить следующие элементы:

- имена полей таблиц;
- литералы;
- операции сравнения;
- арифметические операции;

- логические операции;
- круглые и квадратные скобки.

Если имя поля содержит пробелы, то его заключают в квадратные скобки, в противном случае квадратные скобки необязательны.

Литерал представляет собой значение, заданное явно, например, число, строка или символ. Отметим, что имена переменных в выражении фильтра использовать нельзя. Если в фильтр требуется включить значение переменной или свойства какого-либо компонента, то это значение должно быть преобразовано в строковый тип.

Операции сравнения представляют собой обычные для языка Pascal отношения <, >, =, <=, >= и <>.

Арифметическими являются операции +, -, * и / (сложения, вычитания, умножения и деления соответственно).

В качестве логических операций можно использовать AND, OR и NOT (логическое умножение, сложение и отрицание соответственно).

Круглые скобки применяются для изменения порядка выполнения арифметических и логических операций.

В качестве примеров задания условий фильтрации приведем следующие выражения:

```
Salary <= 2000
Post = 'Лаборант' OR Post = 'Инженер'
```

Первое выражение обеспечивает отбор всех записей, для которых значение поля оклада (Salary) не превышает 2000. Второе выражение обеспечивает отбор записей, поле должности (Post) которых содержит значение Лаборант или Инженер.

Если выражение фильтра не позволяет сформировать сложный критерий фильтрации, то в дополнение к нему можно использовать обработчик события OnFilterRecord.

Для *активизации* и *деактивизации* фильтра применяется свойство Filter типа Boolean. По умолчанию это свойство имеет значение False, и фильтрация выключена. При установке свойства Filtered в значение True фильтрация включается, и в набор данных отбираются записи, которые удовлетворяют фильтру, записанному в свойстве Filter. Если выражение фильтра не задано (по умолчанию), то в набор данных попадают все записи.

Замечание

Активизация фильтра и выполнение фильтрации возможны также на этапе разработки приложения.

Если выражение фильтра содержит ошибки, то при попытке выполнить его генерируется исключение. Если фильтр неактивен (свойство Filtered имеет значение False), то выражение фильтра не анализируется на корректность.

Параметры фильтрации задаются с помощью свойства FilterOptions типа TFilterOptions. Это свойство принадлежит к множественному типу и может принимать комбинации двух значений:

- foCaseInsensitive — регистр букв не учитывается, т. е. при задании фильтра Post = 'Водитель' слова Водитель, ВОДИТЕЛЬ или водитель будут восприняты как одинаковые.

Значение `foCaseInsensitive` нужно отключать, если требуется различать слова, написанные в различных регистрах;

- `foNoPartialCompare` — выполняется проверка на полное соответствие содержимого поля и значения, заданного для поиска. Обычно применяется для строк символов. Если известны только первые символы (или символ) строки, то нужно указать их в выражении фильтра, заменив остальные символы на звездочки (`*`) и выключив значение `foNoPartialCompare`. Например, при выключенном значении `foNoPartialCompare` для фильтра `Post = 'B*'` будут отобраны записи, у которых в поле `Post` содержатся значения `Водитель`, `Вод.`, `Вод-ль` или `Врач`.

По умолчанию все параметры фильтра выключены, и свойство `FilterOptions` имеет значение `[]`.

Рассмотрим в качестве примера обработчики событий формы, используемой для фильтрации записей набора данных по выражению. Вид формы приведен на рис. 19.4.

Управление фильтрацией набора данных выполняется с помощью двух кнопок и поля редактирования. При нажатии кнопки **Фильтровать** (`btnFilter`) фильтр активизируется путем присваивания значения `True` свойству `Filtered` набора данных. Редактор `edtFilter` предназначен для задания выражения фильтра. При активизации фильтра происходит отбор записей, которые удовлетворяют заданному в выражении условию. При нажатии кнопки **Все записи** (`btnAllRecord`) фильтр отключается, при этом показываются все записи.

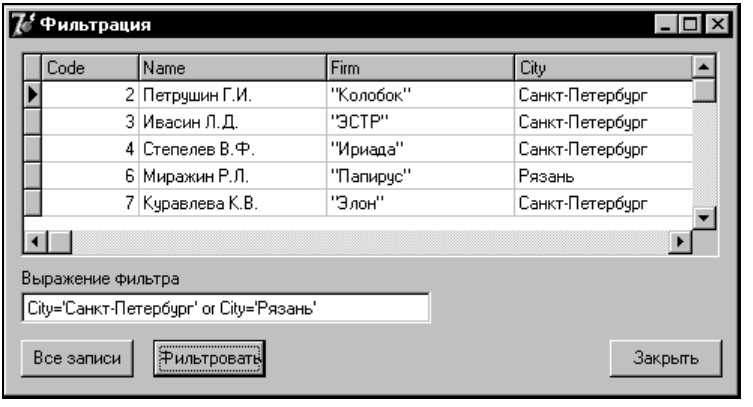


Рис. 19.4. Фильтрация по выражению

Далее приведены три обработчика событий модуля формы `Form1` приложения.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.FilterOptions := [foCaseInsensitive];
  Table1.Filtered := False;
end;

procedure TForm1.btnFilterClick(Sender: TObject);
begin
  Table1.Filtered := True;
```

```
  Table1.Filter := edtFilter.Text;
end;

procedure TForm1.btnAllRecordClick(Sender: TObject);
begin
  Table1.Filtered := False;
end;
```

Включение и выключение фильтра осуществляется через свойство `Filtered`. Фильтрация выполняется без учета регистра букв.

В приведенном примере пользователь должен самостоятельно набирать выражение фильтра. Это предоставляет пользователю достаточно широкие возможности управления фильтрацией, но требует от него знания правил построения выражений.

Часто удобно предоставить пользователю список готовых выражений (шаблонов) для выбора. При этом пользователь получает также возможность редактировать выбранное выражение и корректировать весь список. Такой режим реализуется, например, с помощью компонентов `ComboBox` и `Memo`.

Если набор условий фильтрации ограничен и не изменяется, то пользователь может управлять отбором записей с помощью таких компонентов, как флажки (`CheckBox`) и переключатели (`RadioButton`).

Рассмотрим в качестве примера обработчики событий формы приложения, в которой пользователю предоставлена возможность управлять фильтрацией по двум полям или по выражению, либо совсем отключать фильтрацию (рис. 19.5).

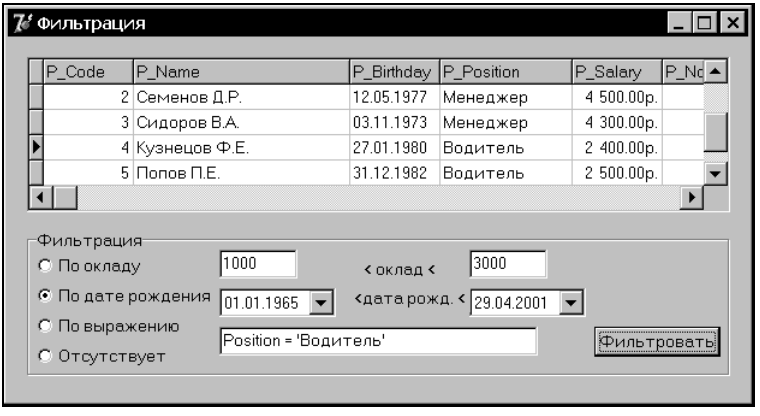


Рис. 19.5. Управление фильтрацией по выражению

Условия фильтрации по полям `Salary` и `BirthDay` заданы в виде двойного неравенства на этапе разработки приложения и при выполнении приложения не могут быть изменены пользователем. Пользователь может задавать в этом неравенстве минимальное и максимальное значения. Фильтрация записей происходит при нажатии кнопки **Фильтровать**. В обработчике события нажатия этой кнопки автоматически формируется выражение фильтра на основании выбранного для фильтрации поля и введенных пользователем значений.

Далее приведены обработчики событий модуля формы `Form1` приложения.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.Filter := '';
  Table1.FilterOptions := [foCaseInsensitive];
  Table1.Filtered := True;
end;

procedure TForm1.btnFilterClick(Sender: TObject);
begin
  // Фильтровать по окладу
  if rbFilterSalary.Checked then Table1.Filter := 'Salary > ' +
     edtSalaryMin.Text + ' AND Salary < ' + edtSalaryMax.Text;
  // Фильтровать по дате рождения
  if rbFilterBirthDay.Checked then Table1.Filter :=
      'BirthDay > ' + DateToStr(dtpBirthDayMin.Date) +
      ' AND BirthDay < ' + DateToStr(dtpBirthDayMax.Date);
  // Фильтровать по выражению
  if rbFilterExpression.Checked then Table1.Filter := edtFilter.Text;
  //  Отключить фильтрацию
  if rbNoFilter.Checked then Table1.Filter := '';
end;
```

Для ввода значений минимальной и максимальной даты применяются два компонента типа `TDateTimePicker` (`dtpBirthDayMin` и `dtpBirthDayMax`), с помощью которых пользователь может выбирать дату. Для ограничения значения оклада использованы однострочные редакторы `edtSalaryMin` и `edtSalaryMax`.

Поскольку в выражении фильтра нельзя использовать имена переменных и свойств компонентов, то при его формировании набранные пользователем значения должны преобразовываться в строковый тип. В приведенном примере это относится к значению даты.

Аналогично задаются и более сложные условия формирования фильтра, в том числе с помощью логических операций `OR` и `NOT`. Кроме того, пользователь может, как и в предыдущем примере, управлять процессом отбора записей с помощью выражения фильтра, которое вводится в редакторе `edtFilter`.

В данном примере, в отличие от предыдущего, при отключении фильтрации фильтр остается активным, однако его выражение очищается.

Приведем еще один пример на фильтрацию — реализуем электронную записную книжку. Для этого совместно с набором данных будем использовать элемент управления вкладками `TabControl`, расположенный на странице **Win32** Палитры компонентов.

Рассмотрим обработчики событий формы приложения, имитирующего записную книжку со списком телефонов. Вид книжки с распечаткой фамилий на букву `A` показан на рис. 19.6.

Список владельцев телефонов хранится в таблице `Notebook.db`, с которой связан набор данных `Table1`. Содержимое списка отображается в компоненте `DBGrid1`. Пользователь

может отобразить все фамилии или вывести фамилии, начинающиеся с определенной буквы.

Для управления отбором записей используется фильтрация. Флажок **Показывать все записи** (`CheckBox1`) активизирует и деактивизирует фильтр. По умолчанию флажок установлен, при этом фильтр неактивен, и отображаются все фамилии. Если флажок `CheckBox1` сброшен, то выполняется фильтрация. Отбор записей осуществляется по первой букве фамилии, которая выбирается с помощью вкладок компонента `TabControl1`.

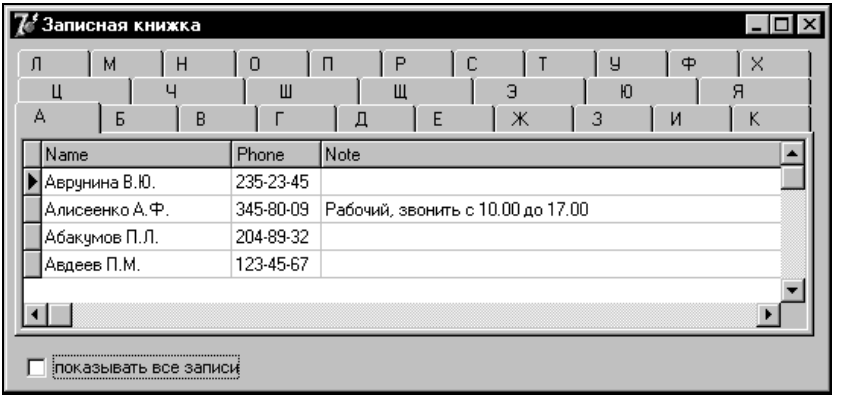


Рис. 19.6. Записная книжка

Далее приведены обработчики событий модуля формы `Form1` приложения.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DBGrid1.Align := alClient;
  // Разрешена фильтрация по частичному совпадению
  // и независимо от регистра букв
  Table1.FilterOptions := [foCaseInsensitive];
  // Отключение фильтрации
  CheckBox1.Checked := True; CheckBox1Click(Sender);
  // Первоначальное формирование выражения фильтра (для буквы А)
  TabControl1Change(Sender);
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked
    then Table1.Filtered := False
    else Table1.Filtered := True;
end;

// Формирование выражения фильтра
// Фильтрация ведется по совпадению первой буквы фамилии
procedure TForm1.TabControl1Change(Sender: TObject);
```

```
begin
  Table1.Filter:='Name =
    ''' + TabControl1.Tabs[TabControl1.TabIndex] + '*''';
end;
```

Здесь при выборе закладки автоматически формируется выражение фильтра, включающее в себя название (букву) выбранной закладки и символ *. Фильтрация выполняется независимо от регистра букв и по частичному совпадению, для чего параметры фильтра содержат значение foCaseInsensitive и не содержат foNoPartialCompare.

При включении фильтрации путем установки свойства Filtered в значение True для каждой отбираемой в набор данных записи генерируется событие OnFilterRecord типа TFilterRecordEvent. В процессе отбора записей набор данных автоматически переводится в режим dsFilter, при этом в нем запрещаются действия, связанные с изменением записей. Тип TFilterRecordEvent описан следующим образом:

```
type TFilterRecordEvent = procedure(DataSet: TDataSet;
                                    var Accept: Boolean) of object;
```

Логический параметр Accept указывает, включать ли в определяемый параметром DataSet набор данных запись, для которой сгенерировано событие OnFilterRecord. По умолчанию параметр Accept имеет значение True, и в набор данных включаются все записи, удовлетворяющие условию фильтра, определяемому свойством Filter. При установке параметра Accept в значение False запись в набор данных не включается.

В обработчике события OnFilterRecord можно определять дополнительные к выражению фильтра условия фильтрации. По своему действию основные и дополнительные условия как бы соединены логической операцией AND, т. е. для отбора записи в набор данных требуется соблюдение обоих условий. В отличие от выражения фильтра, в обработчике события OnFilterRecord можно кодировать любые сколь угодно сложные проверки с помощью средств языка Delphi.

Таким образом, набор данных Table допускает два способа задания условий фильтрации: с помощью выражения фильтра Filter и в обработчике события OnFilterRecord.

Замечание

Обработчик события OnFilterRecord вызывается для каждой записи, считываемой в набор данных, поэтому код его должен быть коротким и оптимизированным, чтобы не снижать производительность приложения в целом.

Событие OnFilterRecord генерируется для каждой записи также при использовании методов поиска FindFirst, FindLast, FindNext и FindPrior *(см. далее в этой главе)*, независимо от значения свойства Filtered.

В случае набора данных Query для отбора записей можно использовать:

- SQL-запрос;
- обработчик события OnFilterRecord;
- выражение фильтра.

Напомним, что для связанных таблиц на отбор записей в набор данных также влияет ограничение, налагаемое отношением "главный — подчиненный" между таблицами БД.

Фильтрация по диапазону

При фильтрации по диапазону в набор данных включаются записи, значения полей которых попадают в заданный диапазон, т. е. условием фильтрации является выражение вида *значение* > *нижняя граница* AND *значение* < *верхняя граница* (вместо операторов сравнения < и > можно использовать операторы <= и >=). Такая фильтрация применяется к наборам данных Table.

Достоинством фильтрации по диапазону является высокая скорость обработки записей. В отличие от фильтрации по выражению, когда последовательно просматриваются все записи таблицы, фильтрация по диапазону ведется индексно-последовательным методом, поэтому этот способ фильтрации применим только для индексированных полей. Индекс поля, диапазон которого задан в качестве критерия для отбора записей, должен быть установлен как текущий с помощью свойства IndexName или IndexFieldNames. Если текущий индекс не установлен, то по умолчанию используется главный индекс.

Для *включения* и *выключения* фильтрации по диапазону применяются методы ApplyRange и CancelRange. Первый из них активизирует фильтр, а второй — деактивизирует. Предварительно для индексного поля (полей), по которому выполняется фильтрация, следует задать диапазон допустимых значений.

Методы SetRangeStart и SetRangeEnd устанавливают нижнюю и верхнюю границы диапазона соответственно. Названные процедуры не имеют параметров, и для задания границ диапазона используется просто инструкция присваивания. При этом методы SetRangeStart и SetRangeEnd переводят набор данных в режим dsSetKey.

Для изменения предварительно установленных границ диапазона предназначены методы EditRangeStart и EditRangeEnd, действие которых аналогично действию методов SetRangeStart и SetRangeEnd соответственно.

Совместно с этими методами используется свойство KeyExclusive типа Boolean, которое определяет, как учитывается заданное граничное значение при анализе записей. Если свойство KeyExclusive имеет значение False (по умолчанию), то записи, у которых значения полей фильтрации совпадают с границами диапазона, включаются в состав набора данных, если же свойство имеет значение True, то такие записи в набор данных не попадают. Свойство KeyExclusive действует отдельно для нижней и верхней границы. Значение этого свойства должно устанавливаться сразу после вызова методов EditRangeStart, EditRangeEnd, SetRangeStart и SetRangeEnd.

Вот как в программе устанавливается нижняя граница диапазона:

```
Table1.SetRangeStart;
Table1.KeyExclusive := True;
Table1.FieldByName('Price').AsFloat := 10.75;
```

Индекс по полю Price должен быть текущим. Записи, содержащие в поле цены (Price) значение 10.75, не входят в отфильтрованный набор данных, поскольку свойству KeyExclusive присвоено значение True. Отметим, что инструкция присваивания, выполняемая после вызова метода SetRangeStart, не меняет значение поля текущей записи, а устанавливает нижнюю границу диапазона.

Когда одна из границ диапазона не задана, то диапазон открыт, т. е. нижняя граница становится равной минимальному возможному, а верхняя граница — максимальному возможному значению этого поля.

Если фильтрация выполняется одновременно по нескольким полям, то после вызова методов `SetRangeStart` или `SetRangeEnd` должны стоять несколько инструкций присваивания, каждая из которых определяет границу по одному полю. Предварительно в качестве текущего должен быть установлен индекс, построенный по этим полям.

Вот пример процедуры, в которой фильтрация осуществляется по нескольким полям:

```
procedure TForm1.btnFilterClick(Sender: TObject);
begin
  with Table1 do begin
     IndexName := 'indNameBirthDay';
     SetRangeStart;
     FieldByName('Name').AsString := 'A';
     FieldByName('BirthDay').AsString := Edit1.Text;
     SetRangeEnd;
     FieldByName('Name').AsString := 'Б';
     ApplyRange;
  end;
end;
```

Здесь для фильтрации одновременно используются поля `Name` и `BirthDay`. Набор данных содержит только записи, соответствующие сотрудникам, которые родились позже даты, введенной в поле редактирования `Edit1`, и с фамилиями, которые начинаются с букв `A` и `Б`. Верхняя граница для даты рождения не задается.

Неправильный ввод даты в поле редактирования `Edit1` является ошибкой и при попытке присвоить это значение полю даты вызывает исключение. Для предотвращения ошибок, связанных с набором даты, можно использовать компонент `DateTimePicker` или проверять вводимые в поле редактирования символы и блокировать неправильный ввод, например, с помощью обработчика события `OnKeyPress`.

В качестве примера рассмотрим обработчики событий модуля формы приложения, обеспечивающего фильтрацию по диапазону (рис. 19.7).

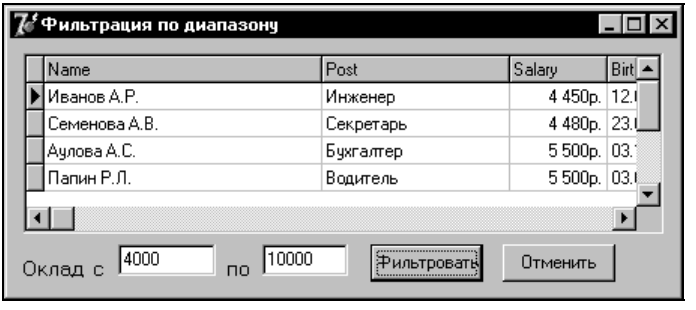


Рис. 19.7. Фильтрация по диапазону

Далее приведены обработчики событий модуля формы `Form1` приложения.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.IndexName := 'indSalary';
end;
```

```
// Фильтрация по полю Salary
// Индекс по этому полю должен быть установлен как текущий
procedure TForm1.btnFilterClick(Sender: TObject);
begin
  with Table1 do begin
     // Установка нижней границы диапазона
     SetRangeStart;
     FieldByName('Salary').AsString := edtRangeMin.Text;
     // Установка верхней границы диапазона
     SetRangeEnd;
     FieldByName('Salary').AsString := edtRangeMax.Text;
     // Включение фильтрации
     ApplyRange;
   end;
end;

// Отключение фильтрации
procedure TForm1.btnNoFilterClick(Sender: TObject);
begin
  Table1.CancelRange;
end;
```

Пользователь может отбирать записи, задавая нижнюю и верхнюю границы оклада. Фильтрация выполняется по полю `Salary`. Предварительно индекс по этому полю должен быть установлен как текущий, в противном случае при попытке фильтрации по диапазону возникнет исключение.

Метод `SetRange` объединяет возможности методов `SetRangeStart`, `SetRangeEnd` и `ApplyRange`. Процедура `SetRange(const StartValues, EndValues: array of const)` позволяет одновременно задать границы диапазона и выполнить фильтрацию. Параметры `StartValues` и `EndValues` являются массивами констант и содержат значения для нижней и верхней границы диапазона соответственно. Если фильтрация выполняется по нескольким полям, то граничные значения для этих полей перечисляются в параметрах `StartValues` и `EndValues` через запятую. Например:

```
// Должен быть установлен текущий индекс,
// построенный по полям должности и оклада
Table1.SetRange(['Преподаватели', 1200], ['Преподаватели',  1500]);
```

В набор данных попадают записи о преподавателях, имеющих оклад от 1200 до 1500 рублей. Поля, по которым выполняется фильтрация и для значений которых устанавливаются границы диапазона, в явном виде не указываются, а являются полями текущего индекса. Соответствующий текущий индекс должен быть установлен до вызова метода `SetRange`.

Для *отмены фильтрации*, выполненной с помощью методов `ApplyRange` или `SetRange`, используется метод `CancelRange`. Другим вариантом отмены предыдущей фильтрации является задание новых границ диапазона, например, методами `SetRangeStart` и `SetRangeEnd`.

Замечание

Если текущий индекс был изменен, то фильтрация по предыдущему индексу перестает действовать, и в наборе данных будут видимы все записи. При возвращении к прежнему текущему индексу фильтрация также не действует, несмотря на то, что диапазон был задан при предыдущей фильтрации.

При выполнении фильтрации по диапазону возможен отбор записей по частичному совпадению значений символьных полей, когда задаются только начальные символы строки. В отличие от фильтрации по выражению, при фильтрации по диапазону отсутствует свойство, аналогичное `FilterOptions` с параметром `foCaseInsensitive`.

В качестве нижней границы указываются символы, являющиеся началом строки, а для получения значения верхней границы можно добавить справа от них строку `яяя`. Как правило, символьные значения в полях для фильтрации не могут содержать несколько идущих подряд букв `я`, а код прописной буквы `я` больше, чем коды других букв, в результате в набор данных попадут записи, поля которых содержат значения, начинающиеся с заданных символов. Если в полях таблицы могут оказаться строки, включающие в себя добавленную строку `яяя`, то можно увеличить количество букв `я` в строке, добавляемой при формировании верхней границы диапазона. При использовании осмысленных слов вероятность такого совпадения близка к нулю, в крайнем случае в отфильтрованном наборе данных может оказаться несколько лишних записей.

Чтобы продемонстрировать фильтрацию с проверкой на частичное совпадение значений, изменим обработчики событий для записной книжки (рис. 19.7) так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DBGrid1.Align := alClient;
  // Главный ключ (по полю Name) используется как текущий индекс
  Table1.IndexFieldNames := 'Name';
  // Первоначальное задание границ диапазона (для буквы А)
  // TabControl1Change(Sender);
  // Отключение фильтрации
  CheckBox1.Checked := True; CheckBox1Click(Sender);
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked
    then Table1.CancelRange
    else TabControl1Change(Sender);
end;

// Задание границ диапазона
// Фильтрация по совпадению первой буквы фамилии
procedure TForm1.TabControl1Change(Sender: TObject);
var Letter: string[1];
begin
  Letter := TabControl1.Tabs[TabControl1.TabIndex];
  Table1.CancelRange;
  Table1.SetRange([Letter], [Letter + 'яяя']);
end;
```

В случае набора данных Query, используя средства SQL, можно отбирать записи по частичному совпадению не только начальных символов строки, но и по вхождению заданных символов в любое место строки.

Навигация с псевдофильтрацией

Система Delphi предоставляет возможность перемещения по набору данных, в котором фильтрация выключена, как по отфильтрованному. Методы FindFirst, FindLast, FindNext и FindPrior перемещают указатель текущей записи соответственно на первую, последнюю, следующую и предыдущую записи, удовлетворяющие условиям фильтрации.

Условия фильтрации должны быть заданы предварительно с помощью выражения фильтра Filter и/или обработчика события OnFilterRecord. При этом фильтр может быть выключен путем установки свойства Filtered в значение False.

На время действия каждого из названных методов набор данных автоматически переводится в режим dsFilter, в результате чего записи *временно* фильтруются, и осуществляется переход на требуемую запись. Связанные с набором данных визуальные компоненты на время фильтрации отображают прежний состав записей набора данных. В результате пользователь не видит хода временной фильтрации, хотя ему запрещено изменять записи, пока набор данных находится в режиме dsFilter.

Таким образом, методы FindFirst, FindLast, FindNext и FindPrior обеспечивают навигацию по записям, удовлетворяющим условиям фильтра, в неотфильтрованном наборе данных.

Замечание

При действии методов FindFirst, FindLast, FindNext и FindPrior используется механизм фильтрации, последовательно перебирающий все записи набора данных, поэтому их применение эффективно только для относительно небольших наборов данных.

На время выполнения названных методов набор данных автоматически переводится в режим dsFilter, поэтому если до вызова любого из методов имелись записи, изменения в которых не были подтверждены, то изменения теряются.

Поиск записей

Поиск записи, удовлетворяющей определенным условиям, означает переход на эту запись. Поиск во многом похож на фильтрацию, т. к. в процессе поиска также выполняется проверка полей записей по некоторым условиям. Отличие заключается в том, что в результате поиска количество записей в наборе данных не изменяется.

При организации поиска записей важное значение имеет наличие индекса для полей, по которым ведется поиск. Индексирование значительно повышает скорость обработки данных, кроме того, ряд методов может работать только с индексированными полями.

Далее мы рассмотрим средства, с помощью которых можно выполнить поиск записей в наборах данных Table и Query. Отметим, что к средствам поиска можно отнести также

методы FindFirst, FindLast, FindNext и FindPrior, осуществляющие переход на записи, удовлетворяющие условиям фильтра.

Поиск в наборах данных

Для поиска записей по полям служат методы Locate и Lookup, причем поля могут быть неиндексированными.

Функция Locate(const KeyFields: String; const KeyValues: Variant; Options: TLocateOptions): Boolean ищет запись с заданными значениями полей. Если удовлетворяющие условиям поиска записи существуют, то указатель текущей записи устанавливается на первую из них. Если запись найдена, функция возвращает значение True, в противном случае — значение False. Список полей, по которым ведется поиск, задается в параметре KeyFields, поля разделяются точкой с запятой. Параметр KeyValues типа Variant указывает значения полей для поиска. Если поиск ведется по одному полю, то параметр содержит одно значение, соответствующее типу поля, заданного для поиска.

Параметр Options позволяет задать значения, которые обычно используются при поиске строк. Этот параметр принадлежит к множественному типу TLocateOptions и принимает комбинации следующих значений:

- loCaseInsensitive (регистр букв не учитывается);
- loPartialKey (допускается частичное совпадение значений).

Отметим, что тип TLocateOptions по сути похож на тип TFilterOptions, определяющий параметры фильтрации по выражению, но значения loPartialKey и foNoPartialCompare имеют противоположное действие: первое из них допускает, а второе запрещает частичное совпадение значений.

Замечание

При наличии у параметра Options значения loPartialKey к нему автоматически добавляется значение loCaseInsensitive.

Пример поиска по одному полю:

```
Table1.Locate('Number', 123, []);
```

Поиск выполняется по полю Number и ищется первая запись, для которой значением этого поля является число 123. Все параметры поиска отключены. Возвращаемый методом Locate результат не анализируется.

При поиске *по нескольким полям* в методе Locate параметр KeyValues является массивом значений типа Variant, в котором содержится несколько элементов. Для приведения к типу вариантного массива используется функция VarArrayOf. Значения разделяются запятыми и заключаются в квадратные скобки, порядок значений должен соответствовать порядку полей параметра KeyFields. Вот пример:

```
Table1.Locate('Name;Post;', VarArrayOf(['П', 'Инженер']),
              [loCaseInsensitive, loPartialKey]);
```

Поиск выполняется по полям Name и Post, ищется первая запись, для которой значение поля фамилии начинается с букв п или П, а значение поля должности содержит строку Инженер. Регистр букв значения не имеет. Результат поиска не анализируется.

Обычно при разработке приложений пользователю предоставляется возможность влиять на процесс поиска с помощью элементов управления, расположенных в форме. При этом действия пользователя по управлению поиском в наборе данных мало чем отличаются от аналогичных действий при выполнении фильтрации.

Замечание

Если имя поля или тип значения заданы неправильно, то при попытке выполнить метод Locate генерируется исключение.

Метод Locate позволяет вести поиск по любым полям, однако если поля индексированы или являются частью некоторого индекса, то соответствующий индекс при поиске используется автоматически. При этом также автоматически записи набора данных сортируются по указанному индексу. В результате, если условиям поиска удовлетворяет несколько записей, то будет найдена и установлена текущей первая в порядке сортировки запись. При использовании другого индекса порядок расположения записей, удовлетворяющих условиям поиска, может измениться, и будет найдена другая запись.

В качестве примера рассмотрим обработчик события, обеспечивающий поиск в наборе данных. Вид формы приложения приведен на рис. 19.8.

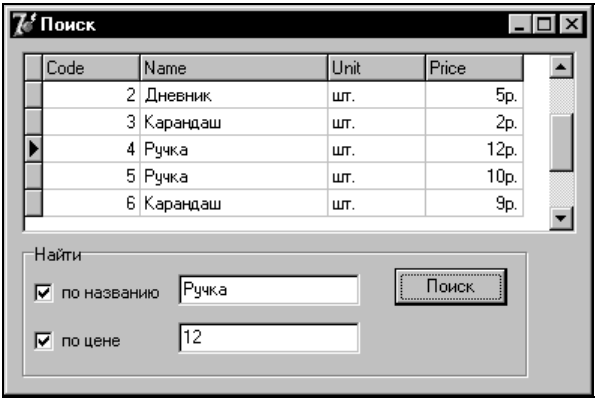


Рис. 19.8. Вид диалогового окна поиска в наборе данных

Пользователь имеет возможность осуществлять поиск по полям названия (Name) и цены (Price) товара. Процесс поиска начинается при нажатии кнопки **Поиск** (btnFind). Далее приведен текст обработчика события нажатия этой кнопки.

```
procedure TForm1.btnFindClick(Sender: TObject);
var KeyFields: String;
    KeyValues: Variant;
      Options: TLocateOptions;
begin
  if not (cbFindName.Checked or cbFindPrice.Checked) then begin
    MessageDlg('Не заданы условия поиска!', mtInformation, [mbOK], 0);
    exit;
  end;
```

```
  // Поиск одновременно по двум полям Name и Price
  if cbFindName.Checked and cbFindPrice.Checked then begin
    KeyFields := 'Name;Price';
    KeyValues := VarArrayOf([edtFindName.Text, edtFindPrice.Text]);
  end
  // Поиск по одному из двух полей
  else begin
    // По полю Name
    if cbFindName.Checked then begin
         KeyFields := 'Name';
         KeyValues := edtFindName.Text;
    end;
    // По полю Price
    if cbFindPrice.Checked then begin
         KeyFields := 'Price';
         KeyValues := edtFindPrice.Text;
    end;
  end;
  // Поиск выполняется независимо от регистра букв
  // с возможностью частичного совпадения
  Options := [loCaseInsensitive, loPartialKey];
  // Запись не найдена
  if not Table1.Locate(KeyFields, KeyValues, Options) then begin
    Beep;
    MessageDlg('Запись не найдена!', mtInformation, [mbOK], 0);
    exit;
  end;
  // Здесь могут быть выполнены действия по обработке найденной записи
end;
```

Включать или не включать поля в поиск — определяется с помощью флажков `cbFindName` и `cbFindPrice`, значения для поиска вводятся в полях редактирования `edtFindName` и `edtFindPrice`.

Если включен параметр `loPartialKey`, то в процессе поиска допускаются частичные совпадения значений. Используя это, можно организовать поиск путем последовательного приближения к требуемой записи. Такой последовательный поиск реализован, например, на страницах **Предметный указатель** и **Поиск** окна справочной системы Windows, где после ввода очередного символа выполняется автоматический переход на подходящую строку. Последовательный поиск можно организовать также с помощью методов `FindNearest`, `SetNearest`, `EditNearest` и `GotoNearest`, предназначенных для поиска с частичным совпадением заданных для поиска значений и значений полей записей.

Рассмотрим в качестве примера обработчики событий формы приложения, используемой для последовательного поиска записей (рис. 19.9).

Далее приведен код обработчиков событий для компонентов формы `Form1` приложения.

```
procedure TForm1.edtSearchChange(Sender: TObject);
var strField: String;
```

```
begin
  // Если режим поиска выключен, то выйти из процедуры
  if not cbSearch.Checked then exit;
  // Выбрать поле для поиска
  case RadioGroup1.ItemIndex of
     0: strField := 'Name';
     1: strField := 'Firm';
     2: strField := 'City';
  end;
  // Выполнить поиск
  Table1.Locate(strField, edtSearch.Text,
              [loCaseInsensitive, loPartialKey]);
end;

procedure TForm1.cbSearchClick(Sender: TObject);
begin
  edtSearchChange(Sender);
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  edtSearchChange(Sender);
end;
```

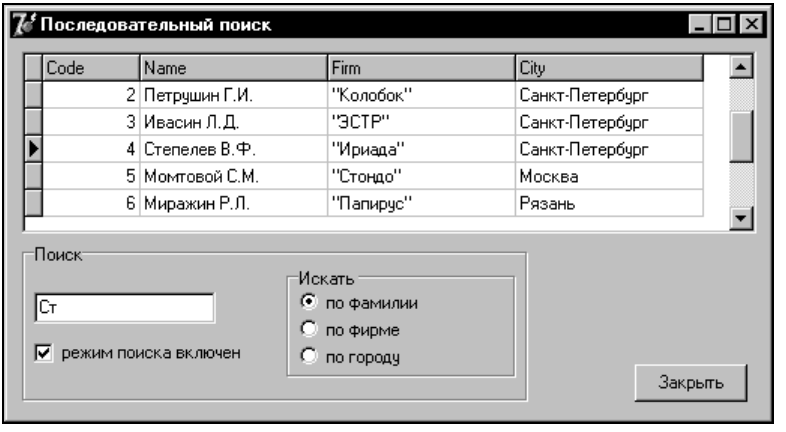


Рис. 19.9. Вид диалогового окна для последовательного поиска

Здесь при установке флажка `cbSearch` включается режим поиска, и указатель текущей записи перемещается на запись, удовлетворяющую условиям поиска. Поле, по которому ведется поиск, выбирается с помощью группы переключателей. Процесс поиска запускается каждый раз при изменении содержимого редактора `edtSearch`.

Для поиска в наборе данных также используется метод `Lookup`, который работает аналогично методу `Locate`. Функция `Lookup(const KeyFields: String; const KeyValues: Variant; const ResultFields: String): Variant` осуществляет поиск записи, удовлетворяющей определенным условиям, но, в отличие от метода `Locate`, не перемещает указатель текущей записи на найденную запись, а считывает информацию из полей

записи. Еще одно различие между двумя методами заключается в том, что метод `Lookup` осуществляет поиск на *точное* соответствие значений для поиска и значений в полях записей с учетом регистра букв.

Параметры `KeyFields` и `KeyValues` имеют такое же назначение, как и в методе `Locate`, и используются аналогичным образом.

В параметре `ResultFields` через точку с запятой перечисляются названия полей, значения которых будут получены в случае успешного поиска. Эти значения считываются из первой найденной записи, удовлетворяющей условиям поиска. Порядок перечисления полей в `ResultFields` может отличаться от порядка полей в наборе данных. Например, если набор данных имеет поля `Code`, `Name`, `Salary` и `Note`, то в `ResultFields` можно задать `Salary` и `Name`.

В случае удачного поиска метод `Lookup` в качестве результата возвращает значение типа `Variant`, размерность которого зависит от списка полей `ResultFields`. Если список содержит одно значение, то метод возвращает значение одного поля, если в списке задано несколько полей, то метод возвращает массив `Variant`, число элементов которого совпадает с числом полей в списке `ResultFields`.

Замечание

Перед использованием результирующие значения элементов массива `Variant` следует самостоятельно преобразовать к требуемому типу, например, `String`, `Real` или `Integer`.

Для работы с массивом `Variant`, число элементов которого заранее неизвестно, предназначены следующие функции:

- `VarIsArray(const V: Variant): Boolean` — проверяет, является ли параметр `V` массивом типа `Variant`;
- `VarArrayLowBound(const A: Variant; Dim: Integer): Integer` — возвращает нижнюю границу массива, заданного параметром `A`, параметр `Dim` определяет размерность массива;
- `VarArrayHighBound(const A: Variant; Dim: Integer): Integer` — возвращает верхнюю границу массива, заданного параметром `A`, параметр `Dim` определяет размерность массива.

При неудачном поиске метод `Lookup` возвращает значение `Null`. Для анализа такого результата можно использовать функцию `VarIsNull(const V: Variant): Boolean`, возвращающую значение `True` при значении параметра `V`, равном `Null`.

Замечание

Не следует путать `Null`, соответствующее пустому (нулевому) значению, и `Nil`, означающее отсутствие значения.

Замечание

Если из найденной записи считываются значения нескольких полей, то даже в случае успешного поиска часть полей может быть не определена и содержать пустые значения. Поэтому при операциях с элементами массива `Variant` следует проверять значение каждого из них на равенство `Null`.

Рассмотрим, как проводится анализ значений полей найденной записи:

```
procedure TForm1.btnFindClick(Sender: TObject);
var KeyFields, ResultFields:    String;
      KeyValues, vrntResult:    Variant;
                 strResult:     String;
begin
  KeyFields := 'Name';
  KeyValues := edtFindName.Text;
  // Порядок полей результата отличается от порядка полей набора данных
  ResultFields := 'Name;Price;Code';
  // Выполнение поиска
  vrntResult := Table1.Lookup(KeyFields, KeyValues, ResultFields);
  // Проверка, успешный ли поиск
  if not VarIsNull(vrntResult) then begin
    // Поиск завершился успешно
    // Анализ поля Name
    if not VarIsNull(vrntResult[0])
      then strResult := 'Название: ' + String(vrntResult[0])
      else strResult := 'Название: -';
    // Анализ поля Price
    if not VarIsNull(vrntResult[1])
      then strResult := strResult + #13#10 + 'Цена: ' +
                        String(vrntResult[1])
      else strResult := strResult + #13#10 + 'Цена: -';
    // Анализ поля Code
    if not VarIsNull(vrntResult[2])
      then strResult := strResult + #13#10 + 'Код: ' +
                        String(vrntResult[2])
      else strResult := strResult + #13#10 + 'Код: -';
    // Вывод значений полей найденной записи
    MessageDlg('Данные найденной записи:' + #13#10 + strResult,
                mtInformation, [mbOK], 0);
  end;
  // Поиск завершился неудачно
  else begin
        Beep;
        MessageDlg('Запись не найдена!', mtInformation, [mbOK], 0);
  end;
end;
```

В примере поиск ведется по полю названия товара (`Name`), а из найденной записи считываются значения полей названия (`Name`), цены (`Price`) и кода (`Code`). Порядок считываемых полей отличается от порядка этих полей в таблице БД. Значение, по которому ищется запись, вводится в редакторе `edtFindName`. После поиска выполняется анализ его успешности, и при положительном результате выводится диалоговое окно с данными полей найденной записи. Если запись не найдена, то выводится соответствующее сообщение. Перед анализом каждого из возвращенных значений проверяется его равенство `Null` (пустое значение). Если поле не пустое, то тип `Variant` преобразовывается в строковый.

Поиск по индексным полям

Для набора данных Table имеются методы, позволяющие вести поиск записей только по индексным полям. Перед вызовом любого из этих методов следует установить в качестве текущего индекс, построенный по используемым для поиска полям. Методы поиска можно разделить на две группы, в первую из которых входят методы FindKey, SetKey, EditKey и GotoKey, предназначенные для поиска на точное соответствие, а другую образуют методы FindNearest, SetNearest, EditNearest и GotoNearest, допускающие только частичное совпадение заданных для поиска значений и значений полей записей.

Метод FindKey(const KeyValues: array of const): Boolean выполняет поиск в наборе данных Table записи, у которой значения полей совпадают со значениями, указанными в параметре KeyValues. Список полей для поиска не задается, а берутся индексные поля в соответствии с текущим индексом. Если поиск завершился успешно, то найденная запись становится текущей, а метод возвращает значение True. При неудачном поиске указатель текущей записи не перемещается, а метод возвращает значение False.

Если заданным условиям поиска соответствует несколько записей, то указатель текущей записи устанавливается на первую из них. Порядок сортировки и, соответственно, порядок расположения записей в наборе данных определяется текущим индексом, по которому и ведется поиск.

Например:

```
procedure TForm1.btnFindClick(Sender: TObject);
begin
  Table1.IndexFieldNames := 'City;Date';
  if not Table1.FindKey([edtFindCity.Text, edtFindDate.Text])
    then ShowMessage('Запись не найдена!');
end;
```

Здесь в наборе данных выполняется поиск первой записи, поля City и Date которой содержат значения, введенные в редакторы edtFindCity и edtFindDate. В случае неудачного поиска пользователю выдается соответствующее сообщение. Перед выполнением поиска текущим устанавливается индекс, построенный по поисковым полям.

Вместо метода FindKey можно вызывать методы SetKey, EditKey и GotoKey, которые применяются совместно. Их использование похоже на применение рассмотренных ранее методов SetRangeStart, EditRangeStart и ApplyRange для фильтрации набора данных по диапазону значений.

Метод SetKey переводит набор данных в режим поиска записи dsSetKey; этот метод вызывается один раз для текущего индекса. Впоследствии для перехода в режим поиска можно вызывать метод EditKey. Если набор данных находится в режиме поиска, значения полей устанавливаются с помощью инструкций присваивания. Метод GotoKey: Boolean выполняет собственно поиск записи, удовлетворяющей заданному условию. Использовать комбинацию названных методов менее удобно, чем метод FindKey. Вот еще один пример, иллюстрирующий точный поиск:

```
procedure TForm1.btnFindClick(Sender: TObject);
begin
  Table1.IndexFieldNames := 'City';
```

```
  // Впоследствии вместо этого метода можно вызывать EditKey
  Table1.SetKey;
  Table1.FieldByName('City').AsString := edtFindCity.Text;
  if not Table1.GotoKey
    then ShowMessage('Запись не найдена!');
end;
```

Приведенная процедура обработки события нажатия кнопки выполняет точный поиск записей по полю города (`City`).

Метод `FindNearest`, в отличие от метода `FindKey`, производит поиск значений полей записей набора данных `Table`, которые только частично совпадают со значениями, заданными для поиска. Сравнение проводится, начиная с первого стоящего в поле символа. Поиск по частичному совпадению можно применять к строкам или к данным других типов, например, целым. Поиск с помощью процедуры `FindNearest(const KeyValues: array of const)` всегда является успешным и перемещает указатель текущей записи на запись, в наибольшей степени отвечающую условиям поиска.

Вместо метода `FindNearest` можно использовать комбинацию методов `SetNearest`, `EditNearest` и `GotoNearest`, работа с которыми аналогична работе с соответствующими методами поиска на точное соответствие значений полей.

Например:

```
procedure TForm1.btnFindClick(Sender: TObject);
begin
  Table1.IndexFieldNames := 'Name';
  Table1.FindNearest(['И']);
end;
```

В приведенной процедуре указатель текущей записи перемещается на запись, поле фамилии которой содержит значение, начинающееся с буквы `И`. Если такой фамилии нет, то будет найдена фамилия, начинающаяся с ближайшей к `И` следующей буквы.

При поиске по нескольким полям текущего индекса используется свойство `KeyFieldCount` типа `Boolean`, указывающее, сколько полей индекса, начиная с первого, участвует в поиске. По умолчанию поиск осуществляется по всем полям текущего индекса.

Модификация набора данных

Модификация (изменение) набора данных представляет собой редактирование, добавление и удаление его записей. Модифицируемость набора данных зависит от различных условий. Разработчик может разрешить или запретить изменение набора данных с помощью соответствующих свойств.

Управлять возможностью изменения набора данных `Table` можно посредством свойства `ReadOnly` типа `Boolean`, при установке которого в значение `True` изменение записей запрещается. По умолчанию свойство `ReadOnly` имеет значение `False`, и набор данных можно модифицировать.

Замечание

Значение свойства `ReadOnly` можно изменять только у закрытого набора данных.

В отличие от многих элементов управления, например, редакторов `Edit` и `Memo`, данный запрет на редактирование относится как к визуальному (пользователем), так и к программному изменению записей набора данных.

Возможность модификации набора данных `Query` определяет свойство `RequestLive` типа `Boolean`. По умолчанию это свойство имеет значение `False`, и набор данных `Query` доступен только для чтения. Чтобы получить разрешение визуально и программно редактировать записи, свойство `RequestLive` нужно установить в значение `True`. Кроме этого свойства, возможность изменения набора `Query` зависит также от содержания SQL-запроса. Например, если при запросе отбираются записи из нескольких таблиц, то набор данных не может быть модифицируемым, и значение `True` свойства `RequestLive` не учитывается.

Для проверки, можно ли изменять набор данных, предназначено свойство `CanModify` типа `Boolean`, действующее при выполнении приложения и доступное только для чтения. Если это свойство имеет значение `True`, то набор данных изменять можно, а если `False`, то изменения в наборе данных запрещены, и любая попытка сделать это визуально или программно вызовет исключение.

Замечание

Можно запретить редактирование отдельных полей набора данных даже в том случае, если он является модифицируемым.

Для программного изменения набора данных вызываются соответствующие методы, например, метод `Edit` редактирования текущей записи или метод `Append` вставки новой записи.

Пользователь редактирует набор данных с помощью визуальных компонентов, например, редактора `DBEdit` или сетки `DBGrid`, управляя ими с помощью мыши и клавиатуры. Набор данных может автоматически переводиться в режимы редактирования или вставки, для этого свойство `AutoEdit` источника данных `DateSource` для визуальных компонентов должно быть установлено в значение `True` (по умолчанию). Если это свойство установить в значение `False`, то пользователь не сможет изменять набор данных с помощью визуальных компонентов.

Замечание

Свойство `AutoEdit` влияет на визуальные компоненты, подключенные к источнику данных `DateSource`, и не оказывает никакого влияния на другие элементы управления, такие как, например, кнопка `Button` или флажок `CheckBox`.

При модификации набора данных для связанного с ним источника данных `DateSource` генерируется событие `OnUpdateData`.

После модификации набора данных возможна ситуация, когда изменения сделаны, но не отображены визуальными компонентами, связанными с этим набором. В таких случаях нужно вызывать метод `Refresh`, который повторно считывает набор данных и тем

самым гарантирует, что визуальные компоненты будут отображать текущие, а не устаревшие данные. При работе в многопользовательской системе рекомендуется периодически вызывать метод `Refresh`, чтобы своевременно учитывать изменения, сделанные другими пользователями.

В однопользовательском приложении обновление набора данных применяется в случае, если с одной таблицей связано *несколько* наборов данных. Например, с таблицей клиентов может быть связан набор данных, с помощью которого выполняется редактирование списка клиентов, а также набор данных, предназначенный для выбора клиента при вводе информации в расходную накладную. Компоненты `Table` обоих наборов могут находиться в разных формах. После редактирования списка клиентов следует обновить оба набора данных, в противном случае возможна ситуация, когда данные о новом клиенте не будут доступны для ввода в накладную. Вот фрагмент кода, в котором проводится обновление набора данных:

```
Table1.Edit;
Table1.FieldByName('Name').AsString:=Edit1.Text;
Table1.Post;
Table1.Refresh;
Form2.Table2.Refresh;
```

В этом примере при программном изменении набора данных `Table1`, находящегося в форме `Form1`, обновляется он сам, а также набор данных `Table2`, расположенный в форме `Form2`. Оба набора связаны с одной и той же физической таблицей. В модуле формы `Form1` должна быть ссылка на модуль формы `Form2`.

Редактирование записей

Редактирование записей заключается в изменении значений их полей. Отредактирована может быть только текущая запись, поэтому перед действиями, связанными с редактированием, обычно выполняются операции поиска и перемещения на требуемую запись. После того как указатель текущей записи установлен на нужную запись, и набор данных находится в режиме просмотра, для редактирования записи следует:

1. Перевести набор данных в режим редактирования.
2. Изменить значения полей записи.
3. Подтвердить сделанные изменения или отказаться от них, в результате чего набор данных снова перейдет в режим просмотра.

Набор данных переводится в режим редактирования вызовом метода `Edit`, при этом возможны такие ситуации:

- если набор данных немодифицируемый, то возбуждается исключение;
- если набор данных уже находился в режиме редактирования или вставки, то никакие действия не выполняются;
- если набор данных пуст, то он переходит в режим вставки.

Если набор данных является модифицируемым и исключение не возбуждается, то при выполнении метода `Edit` производятся следующие действия:

1. Для набора данных вызывается обработчик события `BeforeEdit` типа `TDataSetNotifyEvent`.
2. Из набора данных заново считывается текущая запись и блокируется так, чтобы другие пользователи могли ее читать, но не могли изменять или блокировать. Если операция блокирования завершается неудачно, например, в случае, когда запись уже редактирует другой пользователь, то генерируется исключение, а выполнение метода `Edit` прекращается. Возникшее на этом шаге исключение можно обработать с помощью обработчика события `OnEditError` типа `TDataSetErrorEvent`, если он назначен.
3. Если в записи есть вычисляемые поля, то они пересчитываются.
4. Набор данных переходит в режим редактирования.
5. Для связанного с набором данных источника данных `DataSource` вызывается обработчик события `OnDataChange`.
6. Для набора данных вызывается обработчик события `AfterEdit` типа `TDataSetNotifyEvent`.

Указанные действия осуществляются только для модифицируемого набора данных, поэтому перед вызовом метода `Edit` следует выполнять проверку на возможность редактирования записи (например, путем анализа свойства `CanModify`). Например:

```
if Table1.CanModify then Table1.Edit;
```

Пользователь осуществляет управление набором данных с помощью расположенных в форме элементов как связанных, так и не связанных с набором. Для отдельных визуальных компонентов, связанных с набором данных, переход в режим редактирования происходит различными способами. Например, для компонентов `DBGrid` и `DBEdit` надо сделать двойной щелчок на нужном поле или нажать алфавитно-цифровую клавишу, когда курсор находится в этом поле, а для компонента `DBNavigator` требуется нажать кнопку **Edit Record**. Таким образом, при управлении визуальными компонентами метод `Edit` вызывается пользователем косвенно. В случае, когда набор данных является немодифицируемым, блокировка перехода в режим его редактирования выполняется автоматически и не приводит к ошибке.

Для компонентов управления, *не связанных* с набором данных, например, кнопок `Button` или флажков `CheckBox` программист должен самостоятельно кодировать действия по предотвращению попыток редактирования немодифицируемого набора данных. Вот пример осуществления такой блокировки:

```
procedure TForm1.btnEditClick(Sender: TObject);
begin
  if not Table1.CanModify then begin
    Beep;
    MessageDlg('Редактирование запрещено!', mtInformation, [mbOK], 0);
    exit;
  end;
  Table1.Edit;
end;
```

Здесь переход в режим редактирования осуществляется при нажатии кнопки `btnEdit`, которая может иметь заголовок **Редактировать** или **Edit**. Перед переводом в этот ре-

жим выполняется проверка, можно ли изменять записи набора данных Table1, и если нет, то процедура выдает соответствующее сообщение и завершается.

Другой способ предотвратить редактирование — заблокировать элементы управления, для которых в настоящий момент времени невозможно выполнить обработчики событий. Рассмотрим в качестве примера следующую процедуру:

```
procedure TForm1. cbEditBanClick(Sender: TObject);
begin
  if cbEditBan.Checked then begin
    Table1.Active := False;
    Table1.ReadOnly := True;
    btnEdit.Enabled := False;
    Table1.Active := True;
  end
  else begin
    Table1.Active := False;
    Table1.ReadOnly := False;
    btnEdit.Enabled := True;
    Table1.Active := True;
  end;
end;
```

В ней флажок cbEditBan (с возможным заголовком **Редактирование запрещено**) указывает, допустимо ли изменять записи набора данных Table1. Если этот флажок установлен, то модификация набора данных запрещается, при этом также блокируется кнопка btnEdit вызова метода Edit.

Для блокирования элементов управления можно использовать компонент ActionList, предназначенный для синхронизации элементов управления.

Блокировка попыток пользователя изменить немодифицируемый набор данных должна выполняться также при добавлении и удалении записей.

При выполнении метода Edit непосредственно перед переводом набора данных в режим редактирования возникает событие BeforeEdit, которое можно использовать для проверки возможности перехода в этот режим. Например, при попытке пользователя редактировать запись ему может быть предложено подтвердить свои действия. Для отмены процесса редактирования в обработчике события BeforeEdit можно сгенерировать "тихое" исключение.

При переходе в режим редактирования с помощью кнопки Button проверку его допустимости можно выполнить в обработчике события ее нажатия. Однако обычно удобнее использовать обработчик события BeforeEdit, т. к. оно генерируется при переводе набора данных в режим редактирования любым способом. Вот пример соответствующей процедуры:

```
procedure TForm1.Table1BeforeEdit(DataSet: TDataSet);
begin
  if MessageDlg('Выполнить редактирование?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Abort;
end;
```

После перевода набора данных в режим редактирования можно с помощью инструкций присваивания изменять значения полей текущей записи. При этом нужно учитывать тип поля, выполняя при необходимости операции приведения типов. Например:

```
Table1.FieldByName('City').AsString := Edit1.Text;
Table1.FieldByName('Code').AsInteger := StrToInt(Edit2.Text);
Table1.FieldByName('Price').AsFloat := StrToFloat(Edit3.Text);
```

Перед выполнением приведенных инструкций набор данных `Table1` должен находиться в режиме редактирования или вставки. Если редактор `Edit2` или `Edit3` содержит данные в формате, не соответствующем целому и вещественному числам, то генерируется исключение.

Для проверки, вносились ли изменения в запись, можно проанализировать свойство `Modified` типа `Boolean`. Если свойство имеет значение `True`, то было изменено значение как минимум одного поля текущей записи.

После ввода информации сделанные изменения должны быть или подтверждены, или отменены.

Метод `Post` *записывает модифицированную запись* в таблицу БД, снимает блокировку записи и переводит набор данных в режим просмотра. Если набор данных не находился в режиме редактирования, то вызов метода `Post` приведет к генерации исключения. Перед его выполнением автоматически вызывается обработчик события `BeforePost` типа `TDataSetNotifyEvent`, а сразу после выполнения — обработчик события `AfterPost` типа `TDataSetNotifyEvent`. Используя событие `BeforePost`, можно проверить сделанные изменения и при необходимости отменить их, например, прервав выполнение метода с помощью вызова "тихого" исключения.

Рассмотрим процедуру, в которой осуществляется редактирование записей:

```
procedure TForm1.btnPriceChangeClick(Sender: TObject);
     var bm1:  TBookmark;
    coeff, x:  real;
begin
  // Проверка, является ли набор данных модифицируемым
  if not Table1.CanModify then begin
    MessageDlg('Записи изменять нельзя!', mtError, [mbOK], 0);
    exit;
  end;
  // Получение коэффициента
  try
    coeff := StrToFloat(Edit1.Text);
  except
    MessageDlg('Неправильный коэффициент!' + #13#10 +
               'Повторите ввод.', mtError, [mbOK], 0);
    if Edit1.CanFocus then Edit1.SetFocus;
    exit;
  end;
  // Запоминание позиции текущего указателя
  bm1 := Table1.GetBookmark;
  // Отключение отображения записей визуальными компонентами
  Table1.DisableControls;
```

```
  // Перебор всех записей
  Table1.First;
  while not Table1.EOF do begin
      // Чтение цены из очередной записи
      x := Table1.FieldByName('Price').AsFloat;
      // Пересчет цены
      x := x * Coeff;
      // Изменение цены в текущей записи
      Table1.Edit;
      Table1.FieldByName('Price').AsFloat := x;
      Table1.Post;
      // Переход к следующей записи
      Table1.Next;
  end;
  // Восстановление позиции текущего указателя
  // и отображение записей визуальными компонентами
  if Table1.BookmarkValid(bm1) then Table1.GotoBookmark(bm1);
  if Table1.BookmarkValid(bm1) then Table1.FreeBookmark(bm1);
  Table1.EnableControls;
end;
```

Здесь для всех записей набора данных `Table1` выполняется пересчет поля цены `Price`. Цены умножаются на коэффициент, который введен в редакторе `Edit1`.

Метод `Post` вызывается автоматически при переходе к другой записи с помощью методов `First`, `Last`, `Next` и `Prior`, если набор данных находится в режиме редактирования, и изменения в записях не подтверждены. Поэтому в приведенном примере метод `Post` можно было не вызывать, т. к. сразу после него вызывается метод `Next`. Однако при использовании методов `FindFirst`, `FindLast`, `FindNext` и `FindPrior` неподтвержденные изменения в записях будут потеряны.

Пользователь подтверждает сделанные в записях изменения, управляя соответствующими компонентами, явно или неявно вызывающими метод `Post`. Конкретные действия пользователя зависят от используемых компонентов. Например, при работе с компонентом `DBGrid` изменения подтверждаются (фиксируются) при переходе к другой записи или нажатии клавиши <Enter>, а в компоненте `DBNavigator` можно нажать кнопку **Post Edit** (Подтвердить изменения).

Независимо от способа вызова, метод `Post` может завершиться неудачно, например, если не заданы значения полей, которые не могут быть пустыми, или значение выходит за установленные для него допустимые пределы. В этом случае набор данных обычно возвращается в состояние, которое было до перехода в режим редактирования. При *ошибке выполнения* метода `Post` генерируется событие `OnPostError` типа `TDataSetErrorEvent`. Кодируя обработчик этого события, можно попытаться исправить ошибку.

Метод `Cancel` *отменяет изменения*, выполненные в текущей записи, и возвращает набор данных в режим просмотра. При выполнении метода `Cancel` вызываются обработчики событий `BeforeCancel` и `AfterCancel` типа `TDataSetNotifyEvent`.

Пользователь может отменить сделанные в записях изменения, используя элементы управления компонентов. Например, при работе с сеткой `DBGrid` изменения отменяются

нажатием клавиши <Esc>, а в компоненте `DBNavigator` — нажатием кнопки **Cancel Edit**.

В случае применения механизма транзакций для отмены изменений сразу в нескольких записях можно обратиться к методу `RollBack` класса `TDateBase`.

При *редактировании* текущей записи последовательность инструкций присваивания и вызовов метода `Post` можно заменить вызовом метода `SetFields`.

Процедура `SetFields(const Values: array of const)` устанавливает все или часть значений полей текущей записи. Параметр `Values` содержит массив значений, которые присваиваются этим полям, при этом порядок значений соответствует порядку полей в наборе данных. Кроме того, должны соответствовать типы полей и типы присваиваемых им значений. Если значений в массиве меньше, чем полей в наборе данных, то эти значения присваиваются первым полям, а оставшиеся поля не изменяются. Чтобы не присваивать значение какому-либо полю, в качестве соответствующего значения задается `Nil`, в результате поле не изменяется. Если число значений в массиве превышает число полей, то при выполнении метода `SetFields` генерируется исключение.

Замечание

Повторим еще раз, что значения `Nil` и `Null` не равны друг другу и имеют разный смысл. `Nil` означает отсутствие значения, а `Null` — пустое (нулевое) значение. Если вместо значения `Nil` указать значение `Null`, то поле будет изменено и содержать значение `Null`.

Если для набора данных использовался Редактор полей, то при выполнении метода `SetFields` учитывается порядок полей, установленный с помощью этого Редактора. В противном случае принимается порядок полей, определенный при создании таблицы БД.

Метод `SetFields` удобно использовать для изменения значений нескольких полей. После его выполнения набор данных автоматически возвращается в режим просмотра. Пример использования метода `SetFields` в программе:

```
Table1.Edit;
Table1.SetFields ([Nil, 'Иванов П.О.', Nil, 'Переводчик', 650]);
```

Здесь во второе, четвертое и пятое поля текущей записи набора данных `Table1` заносятся фамилия, должность и оклад соответственно. Значения первого и третьего полей этой записи не изменяются.

Добавление записей

Добавлять записи можно только к модифицируемому набору данных, в противном случае будет сгенерировано исключение.

Для добавления записи нужно выполнить следующие действия:

1. Перевести набор данных в режим вставки.
2. Задать значения полей новой записи.
3. Подтвердить сделанные изменения или отказаться от них, после чего набор данных снова переходит в режим просмотра.

Для добавления записей используются методы `Insert`, `InsertRecord`, `Append` и `AppendRecord`.

Метод `Insert` переводит набор данных в режим вставки и добавляет к нему новую пустую запись. Новая запись вставляется в позицию, на которой находится указатель текущей записи. При необходимости перед вызовом метода `Insert` нужно выполнить перемещение текущего указателя в требуемую позицию набора данных.

После перевода набора данных в режим вставки дальнейшие действия по заданию (изменению) значений полей, подтверждению или отмене сделанных изменений не отличаются от аналогичных действий при редактировании записи. При этом для задания или изменения значений полей используются инструкции присваивания и метод `SetFields`, а для подтверждения или отмены изменений — методы `Post` и `Cancel`. Некоторые поля новой записи могут остаться пустыми, если до подтверждения им не были присвоены значения.

Замечание

При переходе в режим вставки к набору данных добавляется пустая запись, значения полей которой не заданы. Если запись добавляется к подчиненному набору данных, связанному с главным набором связью "главный — подчиненный", то индексные поля автоматически получают корректные значения, и программист может не заботиться об их заполнении.

Рассмотрим следующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.Insert;
  Table1.FieldByName('Name').AsString := Edit1.Text;
  Table1.FieldByName('Group').AsString := Edit2.Text;
  Table1.Post;
end;
```

Здесь в новой записи задаются значения полей фамилии (`Name`) и группы (`Group`), остальные поля остаются пустыми. В этом и последующих примерах предполагается, что набор данных не связан с главным набором данных, и полям не были автоматически присвоены значения как индексным полям.

Часто удобно устанавливать значения сразу нескольких полей с помощью метода `SetFields`. Напомним, что после выполнения этого метода набор данных автоматически возвращается в режим просмотра, и запись считается включенной в набор данных. Если требуется изменить часть полей новой записи, то нужно использовать редактирование, а не вставку. Например, так:

```
procedure TForm1.btnInsertClick(Sender: TObject);
begin
  if not Table1.CanModify then begin
    MessageDlg('Записи изменять нельзя!', mtError, [mbOK], 0);
    exit;
  end;
  Table1.Insert;
  Table1.SetFields([Nil, edtName.Text, Nil,
                    edtPost.Text, IntToStr(edtCode.Text)]);
end;
```

Здесь значения полей новой записи пользователь вводит в редакторах. Первое и третье поля, а также поля с номером больше пяти остаются пустыми. Перед добавлением записи выполняется проверка, можно ли изменять набор данных `Table1`.

Метод `InsertRecord` объединяет функциональность методов `Insert` и `SetFields`, выполняя те же действия, что и их последовательный вызов. Процедура `InsertRecord(const Values: array of const)` вставляет в позицию указателя текущей записи новую запись, задавая значения всех или части ее полей. Например:

```
procedure TForm1.btnInsertClick(Sender: TObject);
begin
  Table1.InsertRecord([Nil, edtName.Text, Nil, edtPost.Text,
                       IntToStr(edtCode.Text)]);
end;
```

Методы `Append` и `AppendRecord` отличаются от методов `Insert` и `InsertRecord` тем, что вставляют запись в конец набора данных, а не в позицию указателя текущей записи.

Пользователь управляет набором данных, в том числе вставкой записи, с помощью элементов управления формы. Для компонента `DBGrid` новая запись добавляется к набору данных при нажатии клавиши <Insert> или при переходе на последнюю запись. Если в форме находится компонент `DBNavigator`, то новая запись добавляется при нажатии кнопки **Insert Record**.

При добавлении новой записи любым методом возникают события `BeforeInsert` и `AfterInsert` типа `TDataSetNotifyEvent`, а также событие `OnNewRecord` типа `TDataSetNotifyEvent`. В обработчиках событий `BeforeInsert` и `OnNewRecord` можно выполнить действия, связанные с проверкой набранных пользователем данных или с заполнением (инициализацией) части полей новой записи. Вот пример соответствующей процедуры:

```
procedure TForm1.Table1NewRecord(DataSet: TDataSet);
begin
  Table1.FieldByName('Unit').AsString := 'штука';
  Table1.FieldByName('NDS').AsString := '20';
end;
```

При утверждении или отмене изменений, связанных с добавлением новой записи, также генерируются события `BeforePost` и `AfterPost` или `BeforeCancel` и `AfterCancel`.

Удаление записей

Удаление текущей записи выполняет метод `Delete`, который работает только с модифицируемым набором данных. В случае успешного удаления записи текущей становится следующая запись, если же удалялась последняя запись, то курсор перемещается на предыдущую запись, которая после удаления становится последней. В отличие от некоторых СУБД, в Delphi удаляемая запись действительно удаляется из набора данных.

Обычно метод `Delete` вызывается для удаления просматриваемой записи, однако с его помощью можно удалить и редактируемую запись. Если набор данных находится в

режиме вставки или поиска, то вызов метода `Delete` аналогичен вызову метода `Cancel`, отменяя соответственно вставку или поиск записи.

Замечание

Если набор данных пуст, то вызов метода `Delete` порождает исключение.

При удалении записи генерируются события `BeforeDelete` и `AfterDelete` типа `TDataSetNotifyEvent`. Используя обработчик события `BeforeDelete`, можно отменить операцию удаления, если не соблюдаются определенные условия.

Если выполнение метода `Delete` приводит к ошибке, то возбуждается исключение, и генерируется событие `OnDeleteError`, в обработчике которого можно выполнить собственный анализ ошибки. Вот небольшая процедура, в которой перед удалением записи запрашивается подтверждение пользователя:

```
procedure TForm1.btnDeleteClick(Sender: TObject);
begin
  if MessageDlg('Удалить запись?',
       mtConfirmation, [mbYes, mbNo], 0) = mrYes then Table1.Delete;
end;
```

Удаление нескольких последовательно расположенных записей имеет особенность, связанную с тем, что при вызове метода `Delete` в цикле по перебору удаляемых записей не нужно вызывать методы, перемещающие указатель текущей записи. После удаления текущей записи указатель автоматически перемещается на соседнюю (обычно следующую) запись. Так можно удалить все записи набора данных:

```
procedure TForm1.btnDeleteAllClick(Sender: TObject);
var n: longint;
begin
  Table1.Last;
  for n := Table1.RecordCount downto 1 do Table1.Delete;
end;
```

В примере перебор записей выполняется с конца набора данных. После удаления текущей записи указатель снова оказывается на последней записи.

Для набора данных `Table` удалить все записи можно также с помощью метода `EmptyTable`, который вызывается в режиме исключительного доступа к таблице БД.

Перед удалением записи часто предварительно выполняется поиск записи (записей), удовлетворяющей заданным условиям. Для отбора группы удаляемых записей используется фильтрация. Метод `Delete` позволяет удалить записи, видимые в наборе данных. Поэтому с помощью фильтрации можно временно оставить в наборе данных записи, которые подлежат удалению, а после удаления фильтрацию отключить.

Пример формы приложения

Обычно визуальные компоненты, предназначенные для редактирования, добавления и удаления записей, группируются на форме и работают взаимосвязанно. Вместе или рядом с этими компонентами часто располагают элементы управления сортировкой,

фильтрацией и поиском. Тем самым для пользователя обеспечивается удобство выполнения различных операций с данными.

Для примера рассмотрим форму (рис. 19.10), с помощью которой можно изменять записи таблицы, содержащей сведения о товарах. В информацию о товаре входят уникальный код записи (поле `Code`), название товара (поле `Name`), единица измерения (поле `Unit`), цена единицы (поле `Price`) и примечание (поле `Note`). Поле названия и поле цены товара обязательно должны быть заполненными, поле кода — автоинкрементное, и его значение при добавлении новой записи формируется автоматически.

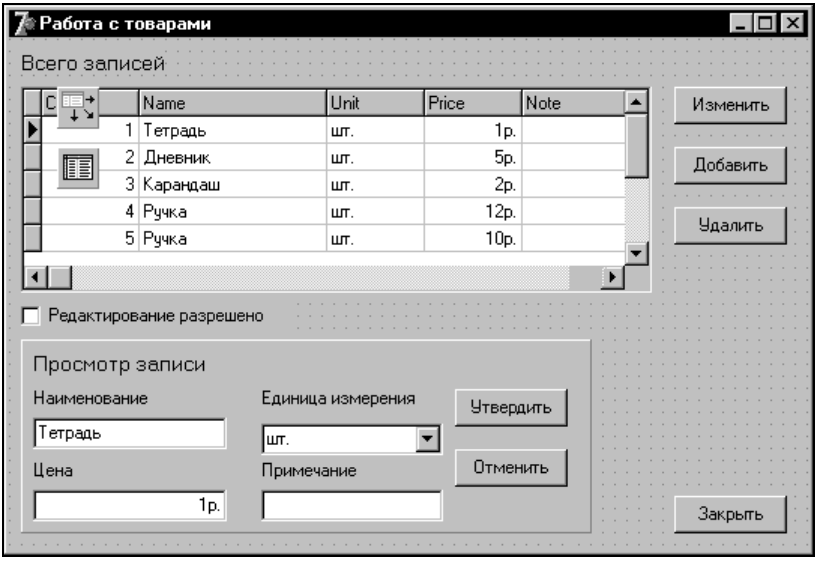


Рис. 19.10. Форма приложения для работы с товарами

В листинге 19.6 приводится код модуля `uChange` формы `Form1` приложения.

Листинг 19.6. Пример формы приложения для работы с товарами

```
unit uChange;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Db, DBTables, Grids, DBGrids, StdCtrls, Mask, DBCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
       DBGrid1: TDBGrid;
    DataSource1: TDataSource;
        Table1: TTable;
       btnEdit: TButton;
     cbCanEdit: TCheckBox;
     pnlChange: TPanel;
        Label1: TLabel;
       dbeName: TDBEdit;
```

```
       dbePrice: TDBEdit;
        dbeNote: TDBEdit;
         Label2: TLabel;
         Label3: TLabel;
         Label4: TLabel;
  lblChangeKind: TLabel;
    btnChangeOK: TButton;
btnChangeCancel: TButton;
       dbcbUnit: TDBComboBox;
       btnClose: TButton;
 lblRecordCount: TLabel;
      btnInsert: TButton;
      btnDelete: TButton;

    procedure StateBrowse(Sender: TObject);
    procedure StateChange(Sender: TObject);
    procedure btnEditClick(Sender: TObject);
    procedure cbCanEditClick(Sender: TObject);
    procedure Table1BeforeEdit(DataSet: TDataSet);
    procedure btnCloseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnChangeOKClick(Sender: TObject);
    procedure btnChangeCancelClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
    procedure btnInsertClick(Sender: TObject);
    procedure Table1BeforePost(DataSet: TDataSet);
    procedure Table1BeforeInsert(DataSet: TDataSet);

  private
    { Private declarations }
  public
    { Public declarations }

  end;

var
  Form1: TForm1;
implementation
{$R *.DFM}
// Режим просмотра

procedure TForm1.StateBrowse(Sender: TObject);
begin
  lblRecordCount.Caption := 'Всего записей — ' +
                            IntToStr(Table1.RecordCount);
  cbCanEditClick(Sender);
  btnChangeOK.Enabled      := False;
  btnChangeCancel.Enabled  := False;
  lblChangeKind.Font.Color := clBlack;
  lblChangeKind.Caption    := 'ПРОСМОТР ЗАПИСИ';
end;
```

```
// Режимы изменения
procedure TForm1.StateChange(Sender: TObject);
begin
  btnEdit.Enabled         := False;
  btnInsert.Enabled       := False;
  btnDelete.Enabled       := False;
  btnChangeOK.Enabled     := True;
  btnChangeCancel.Enabled := True;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  // Исходное состояние элементов управления
  StateBrowse(Sender);
  // Первоначально изменение записей запрещено
  cbCanEdit.Checked := False;
  // Запрет автоматического перехода в режим редактирования
  DataSource1.AutoEdit := False;
  // Формирование списка единиц измерения товара
  dbcbUnit.Items.Clear;
  dbcbUnit.Items.Add('шт.');
  dbcbUnit.Items.Add('упак.');
  dbcbUnit.Items.Add('кор.');
end;

procedure TForm1.cbCanEditClick(Sender: TObject);
var bm1: TBookmark;
begin
  // Запоминание положения текущей записи
  bm1 := Table1.GetBookmark;
  // Отключение отображения изменений данных в визуальных компонентах
  Table1.DisableControls;
  if not cbCanEdit.Checked then begin
    Table1.Active   := False;
    Table1.ReadOnly := True;
    Table1.Active   := True;
    // Блокирование элементов, связанных с переходом
    // в режимы изменения записей
    btnEdit.Enabled   := False;
    btnInsert.Enabled := False;
    btnDelete.Enabled := False;
  end
  else begin
    Table1.Active   := False;
    Table1.ReadOnly := False;
    Table1.Active   := True;
    // Разблокирование элементов, связанных с переходом
    // в режимы изменения записей
    btnEdit.Enabled   := True;
    btnInsert.Enabled := True;
```

```
    // Если набор данных пуст, то удаление записей запрещено
    if Table1.RecordCount > 0
      then btnDelete.Enabled := True
      else btnDelete.Enabled := False;
  end;
  // Возврат к текущей записи
  if Table1.BookmarkValid(bm1) then Table1.GotoBookmark(bm1);
  if Table1.BookmarkValid(bm1) then Table1.FreeBookmark(bm1);
  // Включение отображения изменений данных в визуальных компонентах
  Table1.EnableControls;
end;

procedure TForm1.Table1BeforeEdit(DataSet: TDataSet);
begin
  // Запрос на подтверждение перехода в режим редактирования
  if MessageDlg('Будете редактировать?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Abort;
end;

// Переход в режим редактирования
procedure TForm1.btnEditClick(Sender: TObject);
begin
  Table1.Edit;
  lblChangeKind.Font.Color := clRed;
  lblChangeKind.Caption    := 'РЕДАКТИРОВАНИЕ ЗАПИСИ';
  StateChange(Sender);
  if dbeName.CanFocus then dbeName.SetFocus;
end;

procedure TForm1.Table1BeforeInsert(DataSet: TDataSet);
begin
  // Подтверждение перехода в режим вставки
  if MessageDlg('Добавить запись?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Abort;
end;

// Переход в режим вставки
procedure TForm1.btnInsertClick(Sender: TObject);
begin
  Table1.Insert;
  lblChangeKind.Font.Color := clBlue;
  lblChangeKind.Caption    := 'ВСТАВКА ЗАПИСИ';
  StateChange(Sender);
  if dbeName.CanFocus then dbeName.SetFocus;
end;

// Переход в режим просмотра удаляемой записи
procedure TForm1.btnDeleteClick(Sender: TObject);
begin
  // Подтверждение перехода в режим просмотра удаляемой записи
  if MessageDlg('Удалить запись?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Exit;
```

```
  lblChangeKind.Font.Color := clRed;
  lblChangeKind.Caption    := 'УДАЛЕНИЕ ЗАПИСИ';
  StateChange(Sender);
  if btnChangeCancel.CanFocus then btnChangeCancel.SetFocus;
end;

procedure TForm1.Table1BeforePost(DataSet: TDataSet);
begin
  // Проверка, задано ли значение для названия товара
  if dbeName.Text = '' then begin
    Beep;
    MessageDlg('Не задано название товара!', mtError, [mbOK], 0);
    if dbeName.CanFocus then dbeName.SetFocus;
    Abort;
  end;
  // Проверка, задано ли значение цены товара
  if dbePrice.Text = '' then begin
    Beep;
    MessageDlg('Не задана цена товара!', mtError, [mbOK], 0);
    if dbePrice.CanFocus then dbePrice.SetFocus;
    Abort;
  end;
end;

procedure TForm1.btnChangeOKClick(Sender: TObject);
begin
  // Утверждение изменений в текущей записи (редактируемой или новой)
  // или удаления текущей записи (просматриваемой)
  if Table1.State in [dsEdit, dsInsert]
    then Table1.Post
    else if lblChangeKind.Caption = 'УДАЛЕНИЕ ЗАПИСИ' then Table1.Delete;
  StateBrowse(Sender);
end;

procedure TForm1.btnChangeCancelClick(Sender: TObject);
begin
  // Если набор данных находился в режиме просмотра (при удалении записи),
  // то никаких действий метод Cancel не выполняет
  Table1.Cancel;
  StateBrowse(Sender);
end;

// Закрытие формы
procedure TForm1.btnCloseClick(Sender: TObject);
  begin
  Close;
end;
end.
```

В качестве набора данных используется компонент `Table1`, записи которого отображает сетка `DBGrid1`. Значения полей текущей записи отображаются в компонентах `edtName`,

dbcbUnit, edtPrice и edtNote. Установка свойств, определяющих взаимные отношения компонентов, связанных с набором данных, выполнена при разработке приложения через Инспектор объектов.

Пользователь не может переводить набор данных в режимы изменения с помощью визуальных компонентов, т. к. свойство AutoEdit источника данных DataSource1 установлено в значение False.

Для перевода набора данных в режимы редактирования и вставки служат кнопки **Изменить** (btnEdit) и **Добавить** (btnInsert) соответственно. Переход в эти режимы осуществляется вызовом методов Edit и Insert, после чего название режима отображается в надписи lblChangeKind, блокируются кнопки, связанные с переходом в режимы изменения, и разблокируются кнопки btnChangeOK и btnChangeCancel, позволяющие принять или отменить изменения, сделанные при редактировании или вставке записи. Для подтверждения или отмены изменений вызываются методы Post или Cancel, после чего кнопки подтверждения снова блокируются, а кнопки перехода в режимы изменения разблокируются. Изменение и задание значений полей выполняются с помощью компонентов edtName, dbcbUnit, edtPrice и edtNote.

При переходе в режимы, связанные с изменением записей, пользователю предлагается подтвердить свои действия, что программируется в обработчиках событий OnBeforeEdit и OnBeforeInsert.

Программирование удаления записей отличается от предыдущих действий по изменению набора данных, т. к. режим удаления у набора данных отсутствует. При нажатии кнопки **Удалить** (btnDelete) набор данных остается в режиме просмотра текущей записи. Однако надпись lblChangeKind указывает на переход в режим удаления, и если пользователь подтвердит удаление, нажав кнопку btnChangeOK, то для удаления текущей записи вызывается метод Delete. Таким образом, понятие "режим удаления" относится к форме, а не к набору данных. Запрос на подтверждение перехода в режим удаления кодируется в обработчике события нажатия кнопки btnDelete.

Флажок cbCanEdit с заголовком **Редактирование разрешено** включает и отключает возможность изменения в наборе данных. Управление этой возможностью осуществляется через свойство ReadOnly набора данных. Свойство переключается только при закрытом наборе данных. После нового открытия набора данных указатель текущей записи устанавливается на первую запись, поэтому перед переключением свойства ReadOnly положение указателя текущей записи запоминается с помощью закладки, а после переключения это положение восстанавливается. При разрешении изменений в наборе данных проверяется число его записей, и если набор данных пуст, то кнопка btnDelete блокируется.

Работа со связанными таблицами

Между отдельными таблицами БД может существовать связь, которая организуется через *поля связи* таблиц. Поля связи обязательно должны быть индексированными. Связь между таблицами определяет отношение подчиненности, при котором одна таблица является главной, а вторая — подчиненной. Обычно используется связь "один-комногим", когда одной записи в главной таблице может соответствовать несколько

записей в подчиненной таблице. Такая связь также называется "мастер — детальный" (Master — Detail). После установления связи между таблицами при перемещении в главной таблице текущего указателя на какую-либо запись в подчиненной таблице автоматически становятся доступными записи, у которых значение поля связи равно значению поля связи текущей записи главной таблицы. Такой отбор записей подчиненной таблицы является своего рода фильтрацией.

Для организации связи между таблицами в подчиненной таблице используются следующие свойства, указывающие:

- `MasterSource` — источник данных главной таблицы;
- `IndexName` — текущий индекс подчиненной таблицы;
- `IndexFieldNames` — поле или поля связи текущего индекса подчиненной таблицы;
- `MasterFields` — поле или поля связи индекса главной таблицы.

Работа со связанными таблицами имеет определенные особенности.

- При изменении (редактировании) поля связи может нарушиться связь между записями двух таблиц. Поэтому при редактировании поля связи записи главной таблицы нужно соответственно изменять и значения поля связи всех подчиненных записей.
- При удалении записи главной таблицы нужно удалять и соответствующие ей записи в подчиненной таблице (каскадное удаление).
- При добавлении записи в подчиненную таблицу значение поля связи формируется автоматически по значению поля связи главной таблицы.

Ограничения по изменению полей связи и каскадному удалению записей могут быть наложены на таблицы при их создании, например, в среде программы Database Desktop, или реализовываться программно. Напомним, что эти ограничения ссылочной целостности относятся к так называемым бизнес-правилам — правилам управления БД и поддержания ее в целостном и непротиворечивом состоянии.

Пример приложения

В качестве примера работы со связанными таблицами рассмотрим приложение, предназначенное для автоматизации складского учета.

При организации складского учета используются две таблицы формата Paradox: `Store` для хранения информации о товарах и `Cards` для хранения карточек товара, в которой отмечается движение (приход и расход) каждого товара. Структура таблиц показана в табл. 19.1 и 19.2. В названия полей включены префиксы `S` и `C` (по первым буквам названий таблиц). Такое обозначение помогает при установлении связи между таблицами — из названия поля сразу видно, к какой таблице оно принадлежит.

Между таблицами устанавливается связь "главный — подчиненный", при которой таблица `Store` склада является главной, а таблица `Cards` движения товара — подчиненной (рис. 19.11). Для организации связи в качестве поля связи главной таблицы берется автоинкрементное поле `S_Code` уникального кода товара. По этому полю построен ключ, значение которого автоматически формируется при добавлении новой записи и в пределах таблицы является уникальным. В подчиненной таблице полем связи (внешним ключом) является целочисленное поле `C_Code`, по которому построен индекс.

Таблица 19.1. *Структура таблицы* `Store`

Имя поля	Тип	Размер	Ключевое поле	Примечание
S_Code	+		*	Уникальный код товара. Используется для связи с подчиненной таблицей
S_Name	A	20		Название товара. Требует обязательного заполнения
S_Unit	A	7		Единица измерения. Требует обязательного заполнения
S_Price	$	Не задается		Цена единицы товара. Требует обязательного заполнения
S_Quantity	N	Не задается		Количество товара на складе
S_Note	A	30		Примечание

Таблица 19.2. *Структура таблицы* `Cards`

Имя поля	Тип	Размер	Ключевое поле	Примечание
C_Number	+	Не задается	*	Уникальный код записи о движении товара
C_Code	I	Не задается		Код записи о движении товара, используемый для связи с главной таблицей
C_Move	N	Не задается		Приходное или расходное количество. Требует обязательного заполнения
C_Date	D	Не задается		Дата прихода или расхода

Рис. 19.11. Связь между таблицами `Store` и `Cards`

Приложение для работы со складом включает главную форму `fmStore` (рис. 19.12) и форму `fmInput` ввода данных о новом товаре.

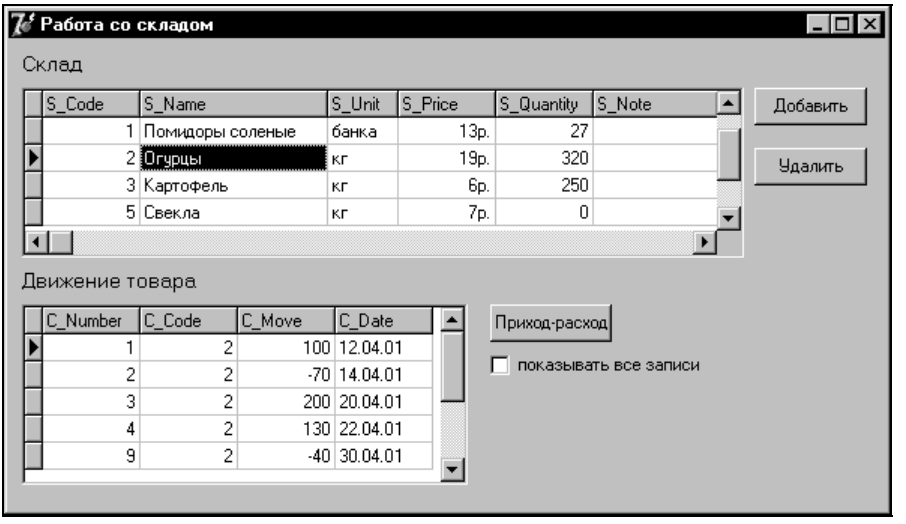


Рис. 19.12. Окно приложения **Работа со складом**

В верхней части главной формы выводится информация о состоянии склада, в нижней части — сведения о движении товара. При выборе в таблице склада записи о товаре в таблице движения товара автоматически отображаются только записи, соответствующие движению именно этого товара. Для наглядности в наборы данных включены все поля таблиц, которые отображаются в компонентах `DBGrid`. При этом названия заголовков столбцов совпадают с названиями полей.

Модификация данных таблиц с помощью компонентов `DBGrid` запрещена, для этого их свойства `AutoEdit` установлены в значение `False`. Для модификации таблиц используются кнопки `Button`, а также отдельная форма `fmInput`. Обработчики событий нажатия кнопок **Добавить** (`btnNew`) и **Удалить** (`btnDelete`) добавляют записи о новом товаре в таблицу склада и удаляют записи о товаре.

При нажатии кнопки `btnNew` выводится в модальном режиме форма `fmInput` (рис. 19.13), содержащая четыре элемента `DBEdit`, которые связаны с полями названия, единицы измерения, цены товара и примечания таблицы `Store`. Связь устанавливается через источник данных `dsStore`, расположенный в главной форме `fmStore`. Чтобы такая связь стала возможной, в модуле `uInput` формы ввода выполнена ссылка на модуль `uStore` главной формы. В свою очередь, в модуле главной формы есть ссылка на форму ввода.

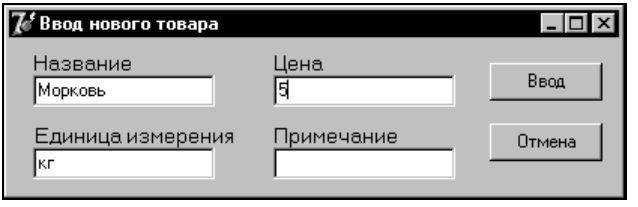


Рис. 19.13. Форма ввода данных о новом товаре

Перед вызовом формы ввода данных о новом товаре в таблицу склада добавляется новая запись, и компоненты-редакторы DBEdit этой формы содержат значения полей (первоначально пустые) новой записи. В процессе ввода пользователь может утвердить ввод, нажав кнопку **OK**, или отменить его, нажав кнопку **Отмена**. После закрытия модальной формы ввода проверяется, какая кнопка была нажата: если **OK**, то сделанные изменения принимаются, в противном случае — нет.

Для удаления записи с данными о товаре следует нажать кнопку btnDelete, после чего выдается запрос на подтверждение операции. В случае подтверждения сначала в цикле удаляются все записи дочерней таблицы с данными о движении этого товара, а затем происходит удаление записи с данными о товаре.

Добавление новой записи в таблицу движения товара выполняется при нажатии кнопки **Приход-расход** (btnMove). При добавлении к таблице движения новой записи поле кода товара, являющееся полем связи, автоматически заполняется правильным значением из текущей записи таблицы склада. В поле даты с помощью инструкции присваивания заносится текущая дата.

Пользователь должен вводить только приходное количество, поэтому для ввода этой информации специальная форма не создавалась, а используется функция InputQuery, позволяющая ввести строковое значение. На практике обычно требуется ввод большего количества данных и применяется форма, построенная таким же образом, как и форма fmInput. Поступление товара (приход) кодируется положительным числом, расход товара — отрицательным числом. После ввода количества товара выполняются преобразование и проверка формата введенного числа. В случае ошибки выдается соответствующее сообщение, и ввод записи отменяется.

После ввода новой записи о движении товара происходит изменение значения поля S_Quantity количества товара в таблице склада.

Для разрыва связи между таблицами используется флажок **показывать все записи** (cbMoveAll). По умолчанию он снят, и связь между таблицами существует. После разрыва связи в таблице движения товара отображаются все записи, независимо от положения текущего указателя в таблице склада. При этом блокируется кнопка btnDelete удаления записей, т. к. при ее нажатии будут удалены все записи о движении товара.

Таким образом, в приложении выполнены следующие действия:

- организована связь между двумя таблицами по полю связи;
- реализовано каскадное удаление записей таблиц;
- запрещено изменение полей связи — пользователь не имеет возможности редактировать их с помощью компонентов DBGrid, а в коде модулей эти поля не затрагиваются.

В листинге 19.7 приведены коды модулей форм приложения. Установка свойств большинства компонентов выполнена в обработчиках событий создания форм приложения.

Листинг 19.7. Пример приложения для автоматизации складского учета

```
// Модуль главной формы
unit uStore;
```

```
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Db, DBTables, ExtCtrls, DBCtrls, Grids, DBGrids, StdCtrls, DBCGrids;
type
  TfmStore = class(TForm)
        dsStore: TDataSource;
     TableStore: TTable;
    DBGridStore: TDBGrid;
         dsCard: TDataSource;
      TableCard: TTable;
     DBGridCard: TDBGrid;
         Label1: TLabel;
         Label2: TLabel;
        btnMove: TButton;
         btnNew: TButton;
      btnDelete: TButton;
      cbMoveAll: TCheckBox;

    procedure FormCreate(Sender: TObject);
    procedure btnNewClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
    procedure btnMoveClick(Sender: TObject);
    procedure cbMoveAllClick(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }

  end;

var
  fmStore: TfmStore;
implementation
uses uInput;
{$R *.DFM}

procedure TfmStore.FormCreate(Sender: TObject);
begin
  dsStore.AutoEdit := False;
  dsCard.AutoEdit := False;
  TableCard.MasterSource := dsStore;
  cbMoveAll.Checked := False;
  cbMoveAllClick(Sender);
end;

procedure TfmStore.btnNewClick(Sender: TObject);
begin
  TableStore.Append;
  if fmInput.ShowModal = mrOK then begin
```

```
    TableStore.FieldByName('S_Quantify').AsFloat := 0;
    TableStore.Post;
  end
  else TableStore.Cancel;
end;

procedure TfmStore.btnDeleteClick(Sender: TObject);
var n: longint;
begin
  if TableStore.RecordCount = 0 then exit;
  if MessageDlg('Удалить запись?', mtConfirmation,
                [mbOK, mbNo], 0) = mrOK then begin
    // Удаление записей в карточке движения товара
    // (с конца набора данных)
    TableCard.Last;
    for n := 1 to TableCard.RecordCount do TableCard.Delete;
    // Удаление карточки движения товара
    TableStore.Delete;
  end;
end;

procedure TfmStore.btnMoveClick(Sender: TObject);
var sMove: string;
    nMove: double;
begin
  if InputQuery('Поступление товара' +
    TableStore.FieldByName('S_Name').AsString,
    'Приход-расход', sMove) then begin
    // Проверка введенного приходного или расходного количества товара
      try
        nMove := StrToFloat(sMove);
      except
        Beep;
        MessageDlg('Неправильно введен приход-расход: ' + sMove,
                   mtError, [mbOK], 0);
        exit;
      end;
      // Добавление новой записи в карточку движения товара
      TableCard.Append;
      // Поле C_Code заполняется автоматически по полю S_Code
      TableCard.FieldByName('C_Move').AsFloat := nMove;
      TableCard.FieldByName('C_Date').AsDateTime := Now;
      TableCard.Post;
      // Пересчет наличного количества товара
      TableStore.Edit;
      TableStore.FieldByName('S_Quantify').AsFloat :=
         TableStore.FieldByName('S_Quantify').AsFloat + nMove;
      TableStore.Post;
  end;
end;
```

```
procedure TfmStore.cbMoveAllClick(Sender: TObject);
begin
  if not cbMoveAll.Checked then begin
    TableCard.IndexName := 'indC_Code';
    TableCard.MasterFields := 'S_Code';
    btnDelete.Enabled := True;
  end
  else begin
    TableCard.IndexName := '';
    TableCard.IndexFieldNames := '';
    TableCard.MasterFields := '';
    btnDelete.Enabled := False;
  end;
end;
end.

// Модуль формы ввода
unit uInput;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Mask, DBCtrls;
type
  TfmInput = class(TForm)
    dbeName: TDBEdit;
    dbeUnit: TDBEdit;
   dbePrice: TDBEdit;
    dbeNote: TDBEdit;
     Label1: TLabel;
     Label2: TLabel;
     Label3: TLabel;
     Label4: TLabel;
     btnOK: TButton;
 btnCancel: TButton;

procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }

  end;

var
  fmInput: TfmInput;
implementation
uses uStore;
{$R *.DFM}
```

```
procedure TfmInput.FormCreate(Sender: TObject);
begin
  dbeName.DataSource    := fmStore.dsStore;
  dbeName.DataField     := 'S_Name';
  dbeUnit.DataSource    := fmStore.dsStore;
  dbeUnit.DataField     := 'S_Unit';
  dbePrice.DataSource   := fmStore.dsStore;
  dbePrice.DataField    := 'S_Price';
  dbeNote.DataSource    := fmStore.dsStore;
  dbeNote.DataField     := 'S_Note';
  btnOK.ModalResult     := mrOK;
  btnCancel.ModalResult := mrCancel;
end;
end.
```

В рассмотренном примере связь между таблицами устанавливалась уже при выполнении приложения. Обычно же таблицы связываются на этапе разработки через Инспектор объектов. При этом для установки свойств `IndexName` и `MasterFields` удобно использовать специальный Редактор полей связи (Field Link Designer), вызываемый двойным щелчком в области значения свойства `MasterFields` в окне Инспектора объектов. В списке **Available Indexes** (Доступные индексы) выбирается индекс подчиненной таблицы, после чего составляющие его поля отображаются в списке **Detail Fields** (Детальные поля). В этом списке необходимо выбрать поле подчиненной таблицы, а в списке **Master Fields** (Главное поле) — поле главной таблицы (рис. 19.14). После нажатия кнопки **Add** выбранные поля связываются между собой, что отображается в списке **Joined Fields** (Связанные поля), например, так: `C_Code -> S_Code`. При этом оба поля пропадают из своих списков. Заполнение свойств `IndexName` и `MasterFields` происходит после закрытия окна при нажатии кнопки **OK**.

Замечание

Перед открытием окна Редактора полей связи необходимо установить значение свойства `MasterSource` подчиненного набора данных `TableCard`, которое должно указывать на источник данных `dsStore` главной таблицы.

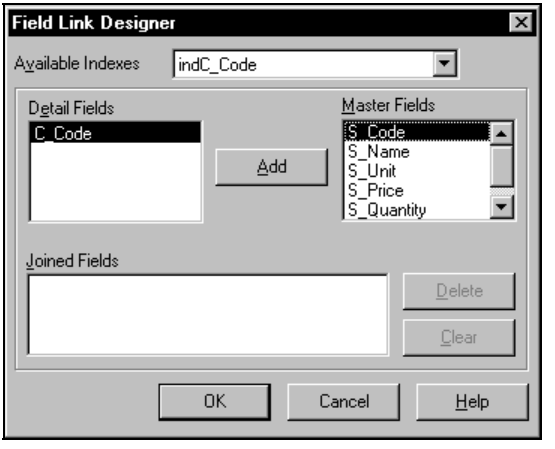


Рис. 19.14. Редактор полей связи

Аналогично можно установить связь между несколькими таблицами, например, таблицей приходной накладной и таблицами поставщиков, покупателей и товаров.

Использование механизма транзакций

При работе с несколькими таблицами БД взаимосвязанные изменения периодически вносятся в разные таблицы. Например, в рассмотренном ранее примере добавлялись новые записи о приходе или расходе товара в таблицу движения товара и соответствующим образом изменялось количество товара в таблице склада.

При возникновении какой-либо ошибки, связанной с записью нового количества товара, новое значение может быть не занесено в соответствующую запись, в результате чего целостность БД нарушится, и она будет содержать некорректные значения. Такая ситуация возможна, например, в случае многопользовательского доступа к БД при редактировании этой записи другим приложением. Поэтому в случае невозможности изменить информацию о количестве товара должно блокироваться и добавление новой записи о движении товара.

Для поддержания целостности БД используется так называемый *механизм транзакций*. Транзакция представляет собой последовательность операций, обычно выполняемую для нескольких наборов данных. Транзакция переводит БД из одного целостного состояния в другое. Чтобы транзакция была успешной, в обязательном порядке должны быть выполнены все операции, предусмотренные в ее рамках (принцип "все или ничего"). В случае возникновения ошибки при выполнении хотя бы одной из операций вся транзакция считается неуспешной, и база данных возвращается в предшествующее транзакции состояние.

Транзакция может быть явной или неявной. *Неявная* транзакция запускается автоматически при модификации набора данных, например, при выполнении методов `Edit`, `Insert`, `Append` и `Delete`. Неявная транзакция утверждается методом `Post` закрепления изменений в наборе данных, а отменяется методом `Cancel`.

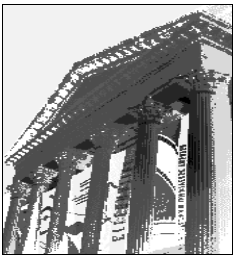
Явная транзакция означает, что программист должен самостоятельно организовывать операции изменения наборов данных. Для реализации механизма явных транзакций Delphi предоставляет специальные методы `StartTransaction`, `Commit` и `Rollback` компонента `DataBase`. Метод `StartTransaction` начинает транзакцию, после него должны располагаться инструкции, составляющие транзакцию. При выполнении операций производится обработка возникающих исключений. Если исключения не возникли, то после выполнения всех операций вызывается метод `Commit`, подтверждающий транзакцию, и все изменения вступают в силу. При возникновении исключения должен вызываться метод `Rollback`, который отменяет транзакцию и действие всех операций в рамках этой транзакции. Более подробно компонент `DataBase` будет рассмотрен в *главе 26*, посвященной работе с удаленными БД. Здесь же мы немного расширим наше приложение по работе со складом, включив в него механизм транзакций путем изменения кода обработчика события нажатия кнопки `btnMove`, как показано далее. Для вызова методов, связанных с запуском и завершением транзакции, в форме размещен компонент `DataBase1`.

```
// Начало транзакции
Database1.StartTransaction;
try
  TableCard.Append;
  TableCard.FieldByName('C_Move').AsFloat := nMove;
  TableCard.FieldByName('C_Date').AsDateTime := Now;
  TableCard.Post;
  TableStore.Edit;
  TableStore.FieldByName('S_Quantity').AsFloat :=
       TableStore.FieldByName('S_Quantity').AsFloat + nMove;
  TableStore.Post;
  // Транзакция выполнена успешно
  // Утвердить изменения
  Database1.Commit;
except
  // Транзакция не выполнена
  // Отказаться от изменений
  Database1.Rollback;
end;
```

В приведенном примере механизм транзакций применяется к связанным таблицам, что в общем случае не обязательно. В одну транзакцию могут быть объединены операции, выполняемые и над отдельными таблицами.

При использовании реляционного способа доступа к данным с помощью инструкций SQL также можно явно управлять транзакциями. Эти возможности рассматриваются в *главе 20*.

ГЛАВА 20



Реляционный доступ к данным с помощью механизма BDE

Реляционный способ доступа к данным основан на операциях с группами записей. Для задания операций используются средства языка структурированных запросов SQL (Structured Query Language), поэтому реляционный способ доступа называют также *SQL*-ориентированным. Для его реализации в приложениях Delphi при использовании механизма BDE в качестве набора данных должны применяться такие компоненты, как `Query` или `StoredProc`, позволяющие выполнить SQL-запрос.

Средства SQL применимы для выполнения операций с локальными и удаленными БД. Наиболее полно преимущества реляционного способа доступа и языка SQL проявляются при работе с удаленными БД. Основным достоинством реляционного способа доступа является небольшая загрузка сети, поскольку передаются только запросы и результат их выполнения.

Применение реляционного способа доступа для локальных БД не дает существенного преимущества, но и в этом случае с помощью SQL-запроса можно:

- формировать состав полей набора данных при выполнении приложения;
- включать в набор данных поля и записи из нескольких таблиц;
- отбирать записи по сложным критериям;
- сортировать набор данных по любому полю, в том числе неиндексированному;
- осуществлять поиск данных по частичному совпадению со значениями в поле.

Замечание

Многие из названных действий неприменимы к набору данных `Table`.

Для компонента `Query` реляционный способ доступа реализуется в случае, когда используются только средства SQL-запросов. Если дополнительно применять методы, ориентированные на операции с отдельными записями, например, `Next` или `Edit`, то будет реализован навигационный способ доступа со всеми его недостатками.

При работе с удаленными БД можно также использовать навигационный способ доступа, но только для небольших сетей, чтобы не создавать большой загрузки.

Основные сведения о языке SQL

Язык SQL ориентирован на выполнение действий с таблицами БД и данными в этих таблицах, а также некоторых вспомогательных действий. В отличие от процедурных языков программирования, в нем нет инструкций управления вычислительным процессом (циклов, переходов, ветвления) и средств ввода/вывода. Составленную на языке SQL программу также называют *SQL-запросом*.

Язык SQL обычно интегрируется в другие средства (оболочку) и используется в интерактивном режиме. Так, в системе управления базами данных, имеющей интерактивный интерфейс, пользователь может работать, ничего не зная о языке SQL и независимо от того, какая БД используется: локальная или удаленная. Такие СУБД, как Microsoft Access, Visual FoxPro или Paradox, сами выполняют действия, связанные с программированием запросов на SQL, предлагая пользователю средства визуального построения запросов, например, Query By Example (QBE) — запрос по образцу.

Поскольку SQL не обладает возможностями полнофункционального языка программирования, а ориентирован на доступ к данным, его часто включают в средства разработки программ. Встроен он и в систему Delphi. При этом для работы с командами SQL предлагаются соответствующие средства и компоненты. В Delphi к таким компонентам относятся наборы данных `Query`, `SQLQuery` и `ADOQuery`.

Различают два вида SQL-запросов: статический и динамический. *Статический* SQL-запрос включается в исходный код на этапе разработки и в процессе выполнения приложения не изменяется. Разработчик может изменить SQL-запрос путем использования параметров, если таковые имеются в его тексте.

Код *динамического* SQL-запроса формируется или изменяется при выполнении приложения. Такие запросы обычно применяются в случае, когда при выполнении запроса требуется учитывать действия пользователя.

Замечание

Принятая нами классификация не является однозначной. Так, в некоторых источниках SQL-запросы с параметрами также относят к разряду динамических.

Язык SQL имеет несколько стандартов, из которых наиболее популярными среди производителей программных продуктов являются стандарты SQL-89 и SQL-92. Стандарт SQL-92, поддерживаемый Американским национальным институтом стандартов (ANSI, American National Standards Institute) и Международной организацией по стандартизации (ISO, International Standard Organization), также называют стандартом ANSI или ANSI/ISO. Вследствие наличия нескольких стандартов и их различных интерпретаций появилось множество диалектов языка SQL, более или менее отличающихся друг от друга.

В языке SQL можно выделить следующие основные подмножества инструкций:

- определения данных;
- обработки данных;
- управления привилегиями (доступом к данным);
- управления транзакциями.

Рассмотрим основные возможности версии языка SQL, используемой в Delphi. Эта версия несколько отличается от стандарта SQL-92, например, в ней нельзя работать с просмотрами и управлять привилегиями.

Особенности языка SQL, используемого для работы с удаленными БД (промышленными СУБД), будут рассмотрены в следующих главах.

В приложениях Delphi для выполнения инструкций SQL, применяя механизм BDE, можно использовать набор данных `Query`. Напомним, что текст SQL-запроса является значением свойства `SQL` компонента `Query` и формируется либо при разработке приложения, либо во время его выполнения. Компонент `Query` обеспечивает выполнение SQL-запроса и получение соответствующего набора данных. Формирование набора данных выполняется при активизации компонента `Query` путем вызова метода `Open` или установкой свойства `Active` в значение `True`. Иногда при отработке SQL-запроса нет необходимости получать набор данных, например, при удалении, вставке или изменении записей. В этом случае предпочтительнее выполнять запрос вызовом метода `ExecSQL`. При работе в сети вызов этого метода производит требуемую модификацию набора данных, не передавая в вызывающее приложение (компьютер) записи набора данных, что существенно снижает нагрузку на сеть.

Кроме того, набрать и выполнить в интерактивном режиме текст SQL-запроса позволяют инструментальные программы, поставляемые вместе с Delphi, например, Database Desktop, SQL Explorer и SQL Builder. Отметим, что первые две программы вызываются одноименными командами меню **Tools** и **Database** соответственно, а визуальный построитель запросов SQL Builder вызывается через контекстное меню компонента `Query`.

Проверка синтаксиса и отработка запроса, встроенного в приложение (чаще всего с помощью компонента `Query`), производятся на этапе выполнения приложения. При наличии синтаксических ошибок в тексте SQL-запроса генерируется исключение, интерпретировать которое порой не просто. Для отладки SQL-запросов удобно использовать программы с развитым интерфейсом, например, Database Desktop. После отладки текст запроса вставляется в разрабатываемое приложение. При таком подходе значительно сокращается время создания запросов и существенно уменьшается вероятность появления динамических ошибок.

В качестве результата выполнения SQL-запроса может возвращаться набор данных, который составляют отобранные с его помощью записи. Этот набор данных называют *результирующим*.

В дальнейшем при описании инструкций языка мы будем опускать несущественные операнды и элементы, используя для обозначения отдельных элементов символы < и > (при программировании не указываются), а необязательные конструкции заключать в квадратные скобки. Для наглядности зарезервированные слова языка SQL будут писаться прописными, а имена — преимущественно строчными буквами. Регистр букв не влияет на интерпретацию инструкций языка. Точка с запятой в конце SQL-инструкций необязательна. Элементы в списках, например имена полей и таблиц, должны быть разделены запятыми.

Имена таблиц и полей (столбцов) заключаются в апострофы или двойные кавычки, например, `"First Name"`. Если имя не содержит пробелы и другие специальные символы, то апострофы можно не указывать.

В SQL-запросе допускаются комментарии, поясняющие текст программы. Комментарий ограничивается символами `/*` и `*/`.

Функции языка

Язык SQL, как и другие языки, предоставляет для использования ряд функций, из которых наиболее употребительны следующие:

- агрегатные, или статистические, функции:
  - `AVG()` (среднее значение);
  - `MAX()` (максимальное значение);
  - `MIN()` (минимальное значение);
  - `SUM()` (сумма);
  - `COUNT()` (количество значений);
  - `COUNT(*)` (количество ненулевых значений);
- функции работы со строками:
  - `UPPER(Str)` (преобразование символов строки `Str` к верхнему регистру);
  - `LOWER(Str)` (преобразование символов строки `Str` к нижнему регистру);
  - `TRIM(Str)` (удаление пробелов в начале и в конце строки `Str`);
  - `SUBSTRING(Str FROM n1 TO n2)` (выделение из строки `Str` подстроки, которая включает в себя символы, начиная с номера (позиции) `n1` и заканчивая номером `n2`);
  - `CAST(<Expression> AS <Type>)` (приведение выражения `Expression` к типу `Type`);
- функция декодирования даты и времени:
  - `EXTRACT(<Элемент> FROM <Выражение>)` (из выражения, содержащего значение даты или времени, извлекается значение, соответствующее указанному элементу); в качестве элемента даты или времени можно указывать значения: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE` или `SECOND`.

Определение данных

Определение данных — это манипулирование целыми таблицами. Сюда включаются операции:

- создания новой таблицы;
- удаления таблицы;
- изменения состава полей таблицы;
- создания и удаления индекса.

Эти действия выполняются с помощью подмножества инструкций определения данных языка SQL.

Создание и удаление таблицы

Для создания таблицы служит инструкция CREATE TABLE, которая имеет следующий формат:

```
CREATE TABLE <Имя таблицы>
       (<Имя поля> <Тип данных>,
        ...
        <Имя поля> <Тип данных>);
```

Обязательными операндами являются имя создаваемой таблицы и имя как минимум одного поля с соответствующим типом данных.

Замечание

В действительности вместо имени таблицы указывается имя главного файла таблицы.

Для локальной таблицы ее формат автоматически определяется по расширению файла: db для таблицы Paradox и dbf для таблицы dBase. Если расширение файла не указано, то тип таблицы определяется драйвером, заданным в BDE для локальных БД *(см. главу 24)*. По умолчанию установлен драйвер Paradox.

Файлы таблицы размещаются в каталоге БД, на который указывает псевдоним БД. Для компонента Query псевдоним задается свойством DatabaseName.

Порядок следования строк с описаниями полей определяет порядок расположения полей создаваемой таблицы. Отметим, что описания полей могут располагаться подряд, а не занимать отдельные строки инструкции.

Типы данных языка SQL и соответствующие им типы данных для таблиц dBase и Paradox приведены в табл. 20.1.

В приведенной таблице N обозначает длину поля в байтах, X — число цифр, Y — число цифр после десятичной точки. Для типа CHARACTER допускается сокращение CHAR. Отметим, что в стандарте SQL-92 число допустимых для полей типов данных намного меньше, например, нет автоинкрементного типа.

Таблица 20.1. *Типы данных для таблиц БД*

SQL	dBase	Paradox
SMALLINT	Number (6,10)	Short
INTEGER	Number (20,4)	Long Integer
DECIMAL(X,Y)	—	BCD
NUMERIC(X,Y)	Number (X,Y)	Number
FLOAT(X,Y)	Number	Float (X,Y)
CHARACTER(N)	Character	Alpha
VARCHAR(N)	Character	Alpha
DATE	Date	Date
BOOLEAN	Logical	Logical
BLOB(N,1)	Memo	Memo

Таблица 20.1 *(окончание)*

SQL	dBase	Paradox
BLOB(N,2)	Binary	Binary
BLOB(N,3)	—	Formatted memo
BLOB(N,4)	OLE	OLE
BLOB(N,5)	—	Graphic
TIME	—	Time
TIMESTAMP	—	Timestamp
MONEY	Number (20,4)	Money
AUTOINC	—	Autoincrement
BYTES(N)	—	Bytes

Пример создания таблицы средствами языка SQL:

```
CREATE TABLE NewTable.dbf
      (Number    INTEGER,
       Name      CHAR(20),
       BirthDay  DATE);
```

В каталоге БД создается новая таблица `NewTable` формата dBase, для которой определены целочисленное поле `Number`, символьное поле `Name` и поле даты `BirthDay`.

Если таблица с заданным именем уже существует, то при выполнении инструкции создания таблицы генерируется исключение.

Для таблицы Paradox можно определить ключ (главный, или первичный), указав описатель `PRIMARY KEY` и перечислив в скобках после него поля, образующие этот ключ. Ключевые поля должны быть в списке полей первыми. Вот пример создания таблицы с построением главного ключа:

```
CREATE TABLE Personnel.db
  (Code    AUTOINC,
   Name     CHAR(20),
   Position CHAR(15),
   Salary   NUMERIC(10,2),
   PRIMARY KEY(Code));
```

Новая таблица `Personnel` имеет формат Paradox, и для нее определены автоинкрементное поле кода `Code`, символьные поля фамилии `Name` и должности `Position`, а также числовое поле оклада `Salary`. По полю кода построен главный ключ.

Для удаления таблицы предназначена инструкция:

```
DROP TABLE <Имя таблицы>;
```

Например:

```
DROP TABLE NewTable.dbf;
```

В результате выполнения этой инструкции с диска удаляются все файлы, относящиеся к таблице с именем NewTable. Если таблица не существует или с ней работает другое приложение, то генерируется исключение.

Изменение состава полей таблицы

Изменение состава полей таблицы заключается в добавлении или удалении полей и приводит к изменению ее структуры, при этом таблицу не должны использовать другие приложения. Изменение состава полей таблицы выполняется инструкцией ALTER TABLE:

```
ALTER TABLE <имя таблицы>
     ADD   <имя поля1> <тип данных1>,
     DROP  <имя поля1>,
     ...
     ADD   <имя поляN> <тип данныхN>,
     DROP  <имя поляN>;
```

Операнд ADD добавляет к таблице новое поле, имя и тип которого задаются так же, как и в инструкции CREATE TABLE, а операнд DROP удаляет из таблицы поле с заданным именем. Операнды ADD и DROP не зависят друг от друга и могут следовать в произвольном порядке.

При попытке удалить отсутствующее поле или добавить поле с существующим именем генерируется исключение.

Пример изменения структуры таблицы:

```
ALTER TABLE Personnel.db
  ADD   Section SMALLINT,
  ADD   Note    CHAR(30),
  DROP  Position;
```

К таблице Personnel добавляются целочисленное поле номера отдела Section и символьное поле примечаний Note, поле Position удаляется.

Создание и удаление индекса

Напомним, что индекс обеспечивает быстрый доступ к данным, хранимым в поле, для которого он создан. Для ускорения операций с таблицей индексными следует делать поля, по которым часто производятся поиск и отбор записей. Индекс создается инструкцией CREATE INDEX следующего формата:

```
CREATE INDEX
      <Имя индекса> ON <Имя таблицы> (<Имя поля1>, ..., [<Имя поляN>]);
```

Одной инструкцией можно создать один индекс, при этом одно поле может входить в состав нескольких индексов. Кроме того, не требуется, чтобы значения составляющих индекс полей были уникальными. При сортировке по индексу записи упорядочиваются по возрастанию значений индексных полей.

С помощью инструкции CREATE INDEX для таблиц dBase создаются индексы, а для таблиц Paradox — вторичные индексы. Напомним, что первичным индексом таблиц Paradox является ключ, описываемый непосредственно при создании таблицы. Исполь-

зование инструкции CREATE INDEX является единственным способом определения индекса для таблиц dBase.

Так можно создать индекс по одному полю:

```
CREATE INDEX
      indName ON Personnel.db (Name)
```

А так по двум:

```
CREATE INDEX
      indNamePosition ON Personnel.db (Name, Position)
```

Для удаления индекса используется инструкция DROP INDEX формата

```
DROP INDEX
      <Имя таблицы>.<Имя индекса>
```

или

```
DROP INDEX
      <Имя таблицы>.PRIMARY
```

Во время удаления индекса таблица не должна использоваться другими приложениями. При выполнении инструкции DROP INDEX можно удалить один индекс, обозначив его составным именем, состоящим из имени таблицы и имени собственно индекса. Если удаляется первичный индекс (ключ) таблицы Paradox, то вместо имени индекса указывается описатель PRIMARY, поскольку главный ключ не имеет имени.

Например, в инструкции

```
DROP INDEX
      "Personnel.db".indNamePosition
```

из таблицы Personnel удаляется индекс indNamePost, созданный по полям Name и Position.

Первичный ключ удаляется так:

```
DROP INDEX
       "Personnel.db".PRIMARY
```

Если удаляемый индекс отсутствует или таблица используется другим приложением, то генерируется исключение.

Отбор данных из таблиц

Отбор данных из таблиц заключается в получении из них полей и записей, удовлетворяющих заданным условиям. Результат выполнения запроса, на основании которого отбираются записи, называют *выборкой*. Данные можно выбирать из одной или нескольких таблиц с помощью инструкции SELECT.

Описание инструкции *SELECT*

Инструкция SELECT — важнейшая инструкция языка SQL. Она служит для отбора записей, удовлетворяющих сложным критериям поиска, и имеет следующий формат:

```
SELECT [DISTINCT] {* | <Список полей>}
      FROM <Список таблиц>
       [WHERE    <Условия отбора>]
       [ORDER BY <Список полей для сортировки>]
       [GROUP BY <Список полей для группирования>]
       [HAVING   <Условия группирования>]
       [UNION    <Вложенная инструкция SELECT>]
```

Результат выполнения SQL-запроса, заданного инструкцией SELECT, представляет собой выборку записей, отвечающих заданным в нем условиям. При рассмотрении инструкции SELECT будем предполагать, что SQL-запрос набран и выполнен с помощью компонента Query. В этом случае результатом выполнения запроса является соответствующий этому компоненту набор данных.

В таком результирующем наборе данных могут быть разрешены или запрещены повторяющиеся записи (т. е. имеющие одинаковые значения всех полей). Этим режимом управляет описатель DISTINCT. Если он отсутствует, то в наборе данных разрешаются повторяющиеся записи.

В инструкцию SELECT обязательно включается список полей и операнд FROM, остальные операнды могут отсутствовать. В списке операнда FROM перечисляются имена таблиц, для которых отбираются записи. Список должен содержать как минимум одну таблицу.

Список полей определяет состав полей результирующего набора данных, эти поля могут принадлежать разным таблицам. В списке должно быть задано хотя бы одно поле. Если в набор данных требуется включить все поля таблицы (таблиц), то вместо перечисления имен полей можно указать символ *. Если список содержит поля нескольких таблиц, то для указания принадлежности поля к той или иной таблице используют составное имя, включающее в себя имя таблицы и имя поля, разделенные точкой: <Имя таблицы>.<Имя поля>.

Операнд WHERE задает условия (критерии) отбора, которым должны удовлетворять записи в результирующем наборе данных. Выражение, описывающее условия отбора, является логическим. Его элементами могут быть имена полей, операции сравнения, арифметические и логические операции, скобки, специальные функции LIKE, NULL, IN и др.

Операнд GROUP BY позволяет выделять группы записей в результирующем наборе данных. Группой являются записи с одинаковыми значениями в полях, перечисленных за операндом GROUP BY. Выделение групп требуется для выполнения групповых операций над записями, например, для определения количества какого-либо товара на складе.

Операнд HAVING действует совместно с операндом GROUP BY и используется для отбора записей внутри групп. Правила записи условий группирования аналогичны правилам формирования условий отбора в операнде WHERE.

Операнд ORDER BY содержит список полей, определяющих порядок сортировки записей результирующего набора данных. По умолчанию сортировка по каждому полю выполняется в порядке возрастания значений; если необходимо задать для поля сортировку по убыванию, то после имени этого поля указывается описатель DESC.

Инструкции SELECT могут иметь сложную структуру и быть вложенными друг в друга. Для объединения инструкций используется операнд UNION, в котором располагается вложенная инструкция SELECT, называемая также *подзапросом*. Результирующий набор данных представляет записи, отобранные с учетом выполнения условий отбора, заданных операндами WHERE обеих инструкций.

Инструкция SELECT используется также внутри других инструкций, например, инструкций модификации записей, обеспечивая для их выполнения требуемый отбор записей.

Управление полями

Управление полями состоит в указании полей таблицы (таблиц), включаемых в результирующий набор данных. Как отмечалось ранее, при отборе всех записей таблицы условия отбора записей не указываются: если вместо списка полей указать *, то в наборе данных оказываются все поля записей. При этом можно не задумываться о названиях полей. Порядок следования полей в наборе данных соответствует порядку физических полей таблицы, определенному при ее создании.

Пример отбора всех полей в таблице:

```
SELECT * FROM Personnel.db
```

В результате выполнения этого запроса из таблицы Personnel в набор данных попадают все поля и все записи, и набор данных имеет вид:

```
Code   Name          Position      Salary
1      Иванов Р.О.   Директор      6700
2      Петров А.П.   Менеджер      5200
3      Семенова И.И. Менеджер      5200
4      Кузнецов П.А. Секретарь     3600
5      Попов А.Л.    Водитель      2400
6      Васин Н.Е.    Водитель      2500
```

При необходимости получения данных из нескольких полей таблицы после слова SELECT через запятую перечисляются в нужном порядке названия этих полей. Порядок полей в наборе данных будет соответствовать порядку полей в списке. Если имя поля указано в списке неоднократно, то в наборе данных оказывается несколько столбцов с одинаковыми именами и данными. Например:

```
SELECT Name, Salary
      FROM Personnel.db
```

В набор данных, формируемый в результате выполнения этого SQL-запроса, включаются поля Name и Salary всех записей из таблицы Personnel. Набор данных получает вид:

```
Name          Salary
Иванов Р.О.   6700
Петров А.П.   5200
Семенова И.И. 5200
Кузнецов П.А. 3600
Попов А.Л.    2400
Васин Н.Е.    2500
```

Можно указать все поля таблицы — такой список аналогичен использованию знака `*`, но отличается тем, что при явном указании имен полей мы можем управлять порядком их следования в наборе данных.

Приведенный достаточно простой запрос демонстрирует определенные преимущества компонента `Query`, использующего возможности языка SQL, по сравнению с компонентом `Table`. Для компонента `Table` изменить порядок или ограничить состав полей набора данных можно только на этапе разработки приложения, задав с помощью Редактора полей статические поля. Компонент `Query` позволяет осуществлять эти операции и для динамических полей, что достигается небольшим изменением SQL-запроса как при разработке, так и при выполнении приложения.

Кроме физических полей таблиц, в набор данных можно включать *вычисляемые поля*. Для получения вычисляемого поля в списке полей указывается не имя этого поля, а выражение, по которому рассчитывается его значение. Выражение составляется по обычным правилам, при этом в него могут не входить имена полей. Например:

```
SELECT "- " || Name, Salary, Salary * 1.1
     FROM Personnel;
```

Для сотрудников организации (таблица `Personnel`) выводятся старое значение оклада и новое, увеличенное на 10%. К каждой фамилии с помощью операции конкатенации (`||`) добавляются дефис и пробел. Результирующий набор будет таким:

```
Name            Salary  Salary * 1.1
- Иванов Р.О.   6700    7370
- Петров А.П.   5200    5720
- Семенова И.И. 5200    5720
- Кузнецов П.А. 3600    3960
- Попов А.Л.    2400    2640
- Васин Н.Е.    2500    2750
```

Записи могут иметь одинаковые значения некоторых полей. Для того чтобы включить в набор данных только записи с *уникальными* (неповторяющимися) значениями, перед списком полей указывается описатель `DISTINCT`. Например:

```
SELECT DISTINCT Position
      FROM Personnel
```

Сформированный набор данных содержит список занимаемых штатных должностей организации. Записи выбираются из таблицы сотрудников организации, при этом в набор данных каждая должность включается только один раз (поле `Position`):

```
Position
Директор
Менеджер
Секретарь
Водитель
```

А так задается выборка записей с уникальными значениями двух полей:

```
SELECT DISTINCT Position, Salary
      FROM Personnel
```

Сформированный набор данных содержит список занимаемых штатных должностей и окладов. В набор данных попадают записи с уникальной комбинацией значений двух полей (`Position` и `Salary`):

```
Position      Salary
Директор      6700
Менеджер      5200
Секретарь     3600
Водитель      2400
Водитель      2500
```

Еще одним достоинством языка SQL является простота объединения в результирующем наборе данных, содержащихся в *нескольких таблицах*. Для этого после слова `FROM` перечисляются имена таблиц, из записей которых формируется набор данных. Такое использование данных из различных таблиц называется *соединением*.

Пример запроса на отбор записей из двух таблиц:

```
SELECT * FROM Personnel, Info
```

или

```
SELECT Personnel.*, Info.* FROM Personnel, Info
```

Результирующий набор данных состоит из всех полей и всех записей таблиц `Personnel` и `Info`. Первыми располагаются поля первой таблицы, далее следуют поля второй таблицы. Имена полей набора данных являются составными и включают в себя имена таблиц.

При выполнении запроса к нескольким таблицам в набор данных отбираются записи этих таблиц, удовлетворяющие заданным условиям. В данном случае условия отбора не заданы, поэтому в набор данных попадают все записи из обеих таблиц. Обычно таким образом отбираются записи из таблиц, связанных по определенным полям. Если же сформировать набор данных из таблиц, содержащих не связанные между собой данные, например, список сотрудников организации и список товаров на складе, то получившиеся записи могут содержать такие поля, как фамилия сотрудника и цена товара, что вряд ли имеет какой-либо смысл.

Еще один пример на отбор полей из разных таблиц:

```
SELECT Personnel.Name, Info.I_Code
      FROM Personnel, Info
```

В случае, если имя таблицы, приводимое в операнде `FROM`, указывает формат файла таблицы, то в обозначении поля имя этой таблицы заключается в кавычки, например, `"Personnel.db".Name`.

Вместо имени таблицы в тексте SQL-запроса можно задать *псевдоним*, упрощающий указание имени таблицы, например, при обозначении ее полей. После определения псевдонима его можно использовать вместо имени таблицы. Псевдоним задается с помощью описателя `AS`, указываемого после имени таблицы. Например:

```
SELECT P.Name, I.I_Code
      FROM "Personnel.db" As P, "Info.db" As I
```

Для таблицы `Personnel` определен псевдоним `P`, а для таблицы `Info` — псевдоним `I`, которые используются при обозначении полей этих таблиц. Отметим, что в данном примере использование псевдонимов встречается раньше, чем их определение. Язык SQL не является алгоритмическим, и в нем допускается такой порядок описания и использования псевдонимов.

Замечание

Псевдоним таблицы применяется в инструкции SQL вместо имени таблицы и никак не связан с псевдонимом БД, который используется для определения значения свойства `DatabaseName` набора данных `Query` и задает расположение БД.

Если в составе таблиц, из которых отбираются записи, имя некоторого поля является уникальным, то имя содержащей это поле таблицы можно не указывать. Например:

```
SELECT Name, I_Code
  FROM Personnel, Info
```

Имена полей `Name` и `I_Code` уникальны для таблиц `Personnel` и `Info`, поэтому имена таблиц в обозначении полей можно опустить.

Простое условие отбора записей

В предыдущем разделе в набор данных попадали все записи из указанных таблиц, при этом разработчик мог управлять составом полей этих записей. На практике набор данных обычно ограничивается записями, удовлетворяющими каким-либо определенным условиям (критериям) отбора, задаваемым с помощью операнда `WHERE`. Критерий отбора может варьироваться от простейшего, в котором сравниваются два значения, до сложного, когда учитывается много факторов. При этом в набор данных попадают записи, для которых выполняется условие отбора.

Критерий отбора представляет собой логическое выражение, в котором можно использовать операции, перечисленные далее.

- Сравнение; возможные операторы:
  - `=` (равно);
  - `>` (больше);
  - `<` (меньше);
  - `>=` (больше или равно);
  - `<=` (меньше или равно);
  - `<>` или `!=` (не равно);
  - `!>` (не больше);
  - `!<` (не меньше).
- `LIKE` (сравнение по шаблону).
- `IS NULL` (проверка на нулевое значение).
- `IN` (проверка на вхождение).
- `BETWEEN` (проверка на вхождение в диапазон).

В простом критерии отбора используется одна операция. Для операций сравнения и сравнения по шаблону критерий отбора имеет следующий формат:

```
<Выражение1> <Оператор> <Выражение2>
```

Выражение состоит из имен полей, функций, констант, значений, знаков операций и круглых скобок. В простейшем случае — из имени поля или значения. Например:

```
SELECT Name
      FROM Personnel
      WHERE Salary >= 4000;
```

Эта инструкция задает получение следующего списка сотрудников, имеющих оклад не менее 4000 (рублей):

```
Name
Иванов Р.О.
Петров А.П.
Семенова И.И.
```

Пример отбора записей по значениям символьного поля:

```
SELECT Name
      FROM Personnel
      WHERE Position = 'Водитель';
```

В результате получим выборку с фамилиями водителей:

```
Name
Попов А.Л.
Васин Н.Е.
```

Поля, входящие в набор данных, и поля, используемые в критерии отбора, могут отличаться друг от друга. Так, в приведенном примере в наборе данных присутствует поле фамилии `Name`, в то время как в критерии отбора записей используется поле оклада `Salary`.

В последнем примере в операции сравнения учитывается регистр символов, и слова `Водитель` и `водитель` не равны друг другу. Поэтому различия в регистре символов или наличие ведущих и конечных пробелов приводят к ошибкам отбора записей. В данной ситуации критерий отбора лучше записать в следующем виде:

```
WHERE UPPER(TRIM(Post)) = 'ВОДИТЕЛЬ'
```

Функция `TRIM` удаляет из строкового значения ведущие и конечные пробелы, а функция `UPPER` приводит символы полученной строки к верхнему регистру. В результате значение должности `водитель` независимо от наличия ведущих и конечных пробелов, а также регистра букв будет приведено к значению `ВОДИТЕЛЬ`.

Для сравнения строк вместо операций `=`, `!=` и `<>` можно использовать операцию `LIKE`, выполняющую сравнение по частичному совпадению. Частичное совпадение значений целесообразно проверять, например, в случаях, когда известны только начальная часть фамилии или названия предмета. Вот образец соответствующего запроса:

```
SELECT Name
      FROM Personnel
      WHERE Name LIKE "Ав"
```

Здесь мы получаем список фамилий, начинающихся на `Ав`.

В выражениях операции `LIKE` допускается использование *шаблона*, в котором разрешены все алфавитно-цифровые символы (с учетом регистра).

При этом два символа имеют специальное назначение:

- `%` (замещение любого количества символов, в том числе и нулевого);
- `_` (замещение одного символа).

С помощью шаблона можно выполнить проверку на частичное совпадение не только начальных символов строки, но и найти вхождение заданного фрагмента в любую часть строкового значения. Например:

```
SELECT Name
      FROM Goods
      WHERE Name LIKE "%" || "ка" || "%"
```

В приведенном запросе происходит отбор всех товаров, в названия которых входят символы `ка`. Набор данных, полученный при таком отборе, может иметь вид:

```
Name
ручка
карандаш
замазка
```

Перед операцией `LIKE` можно использовать описатель `NOT`, который изменяет результат выполнения операции на противоположное значение и проверяет значения выражений на несовпадение.

Для проверки нулевого значения выражения служит операция `IS NULL`, которая имеет следующий формат:

```
<Выражение> IS [NOT] NULL
```

Так, в запросе:

```
SELECT *
      FROM Store
      WHERE S_Price IS NULL
```

выполняется отбор всех полей записей таблицы склада (`Store`), для которых не определена цена (`S_Price`) товара.

Проверка на *вхождение* значения выражения *в список* выполняется с помощью операции `IN` следующего формата:

```
<Выражение> [NOT] IN <Список значений>
```

Эту операцию удобно использовать, если выражение может принимать относительно небольшое количество различных значений. Вот пример соответствующего запроса:

```
SELECT Name, Salary
      FROM Personnel
      WHERE LOWER(Post) IN ("менеджер", "водитель")
```

В результате его выполнения получим выборку фамилий и окладов всех менеджеров и водителей:

```
Name            Salary
Петров А.П.     5200
Семенова И.И.   5200
```

```
Попов А.Л.      2400
Васин Н.Е.      2500
```

Операция BETWEEN выполняет проверку *вхождения* значения *в диапазон* и имеет формат:

```
<Выражение> [NOT] BETWEEN
      <Минимальное значение> AND <Максимальное значение>
```

При использовании этой операции в набор данных включаются записи, для которых значение выражения больше или равно минимальному, а также меньше или равно максимальному значениям.

Замечание

Описатель NOT изменяет значение результата операции на противоположное.

Рассмотрим запрос:

```
SELECT *
      FROM Cards
      WHERE C_Date BETWEEN "21.5.02" AND "27.5.2002"
```

В результате его выполнения получим набор записей, для которых дата (поле C_Date) находится в диапазоне с 21 по 27 мая 2002 года:

```
C_Number   C_Code   C_Move   C_Date
3          3        -150     22.05.02
4          3        -30      22.05.02
5          3        270      24.05.02
6          4        200      27.05.02
```

Значение даты заключается в кавычки, в качестве разделителя используется точка. В некоторых других реализациях языка SQL в качестве разделителей допускается использовать символы — и /. Значение года можно задавать как двумя (21.5.02), так и четырьмя (27.5.2002) цифрами. В значениях дня и месяца можно опускать незначащий ноль.

Сложные критерии отбора записей

При отборе можно использовать несколько операций, задавая тем самым сложные критерии отбора записей.

Сложный критерий (логическое выражение) состоит из:

- простых условий;
- логических операций:
  - AND (логическое И);
  - OR (логическое ИЛИ);
  - NOT (логическое НЕ);
- круглых скобок.

Замечание

В языке SQL приоритет операций сравнения выше приоритета логических операций. Для изменения порядка выполнения операций используются круглые скобки.

Пример запроса со сложным критерием отбора:

```
SELECT Cards.*
      FROM Store,Cards
      WHERE (S_Quantity > 250) AND
            (C_Date BETWEEN "21.5.2002" AND "27.5.2002")
```

Здесь происходит отбор всех записей о движении товара, количество которого на складе превышает 250 единиц, в течение заданного периода. В данном логическом выражении простые условия заключены в скобки, что не обязательно, т. к. приоритет операций сравнения выше приоритета логических операций. Набор данных, полученный при таком отборе записей, может иметь вид:

```
C_Number   C_Code   C_Move    C_Date
5          3        270       25.05.02
6          4        200       26.05.02
5          3        270       27.05.02
6          4        200       27.05.02
```

Отметим, что рассмотренные выше операции `BETWEEN` и `IN` также реализуют логические операции: `BETWEEN` — логическое `И`, а `IN` — логическое `ИЛИ`.

Группирование записей

Записи набора данных могут быть сгруппированы по некоторому признаку. Группу образуют записи с одинаковыми значениями в полях, перечисленных в списке операнда `GROUP BY`. При группировании записей их проще анализировать и обрабатывать, например, с помощью статистических функций.

Группирование записей автоматически исключает повтор значений в полях, заданных для группирования, т. к. записи с совпадающими значениями этих полей объединяются в одну группу.

Пример запроса с группированием записей:

```
SELECT C_Date, COUNT(C_Date)
      FROM Cards
      WHERE C_Date BETWEEN "01.06.2002" AND "03.06.2002"
      GROUP BY C_Date
```

Для каждой даты из указанного периода выводится количество записей, в которых она встречается. Если не выполнить группирование, то в набор данных попадут все записи, а при использовании группирования все даты для полученного набора данных уникальны. Функция `COUNT` выводит для каждой группы (сформированной по полю даты) число записей в группе. Полученный набор данных может иметь следующий вид:

```
C_Date       COUNT(C_Date)
01.06.02     20
02.06.02     17
03.06.02     8
```

В этом примере поле даты C_Date использовано в инструкции SELECT трижды, в общем случае можно использовать различающиеся поля.

Совместно с операндом GROUP BY можно использовать операнд HAVING, с помощью которого задаются дополнительные условия группирования записей.

Рассмотрим пример запроса:

```
SELECT C_Date, COUNT(C_Date)
      FROM Cards
      GROUP BY C_Date
      HAVING COUNT(C_Date) > 50
```

Здесь отбираются данные для дат, когда движение товара было интенсивным — общее число записей в соответствующей группе превышало 50. Так как не указан ограничивающий период времени, при отборе в набор данных будут учитываться записи таблицы Cards с любыми датами.

Сортировка записей

Как уже говорилось, сортировка представляет собой упорядочивание записей по возрастанию или убыванию значений полей. Список полей, по которым выполняется сортировка, указывается в операнде ORDER BY. Порядок полей в этом операнде определяет порядок сортировки: сначала записи упорядочиваются по значениям поля, указанного в этом списке первым, затем записи, имеющие одинаковое значение первого поля, упорядочиваются по второму полю и т. д.

Поля в списке обозначаются именами или номерами, которые соответствуют номерам в списке полей после слова SELECT.

По умолчанию сортировка происходит в порядке возрастания значений полей. Для указания обратного порядка сортировки по какому-либо полю нужно указать после имени этого поля описатель DESC.

Замечание

В отличие от набора данных Table, средствами языка SQL можно выполнять сортировку для набора данных Query и по неиндексированным полям. Однако по индексированным полям таблицы сортировка выполняется быстрее. При этом состав полей индекса должен соответствовать списку полей, указанных в операнде ORDER BY.

Пример запроса на сортировку записей:

```
SELECT * FROM Personnel.db
      ORDER BY Name
```

Сортировка записей задана по полю Name. Полученный набор данных может иметь вид:

```
Code    Name           Position       Salary
6       Васин Н.Е.     Водитель       2500
1       Иванов Р.О.    Директор       6700
4       Кузнецов П.А.  Секретарь      3600
2       Петров А.П.    Менеджер       5200
5       Попов А.Л.     Водитель       2400
3       Семенова И.И.  Менеджер       5200
```

Еще один пример запроса на сортировку — на этот раз по двум полям:

```
SELECT Name, Position, Salary
      FROM Personnel.db
      ORDER BY Position, Salary DESC
```

или

```
SELECT Name, Position, Salary
      FROM Personnel.db
      ORDER BY 2, 3 DESC
```

В набор данных входят поля `Name`, `Position` и `Salary` всех записей. Записи отсортированы по полям `Position` и `Salary`, при этом по полю `Salary` упорядочивание выполняется в порядке убывания значений. Полученный набор данных будет таким:

```
Name           Position       Salary
Васин Н.Е.     Водитель       2500
Попов А.Л.     Водитель       2400
Иванов Р.О.    Директор       6700
Петров А.П.    Менеджер       5200
Семенова И.И.  Менеджер       5200
Кузнецов П.А.  Секретарь      3600
```

Если по полям `Position` и `Salary` построен индекс, то операции с набором данных будут выполняться быстрее.

При разработке приложения управление сортировкой осуществляется посредством различных элементов формы, например, кнопок и переключателей.

Рассмотрим следующий пример. Пользователь управляет сортировкой с помощью двух групп переключателей: в первой задается вид, во второй — направление сортировки. Сортировка выполняется после нажатия кнопки **Отсортировать** (`btnSort`). На рис. 20.1 показан вид формы на этапе проектирования.

В листинге 20.1 приводится код обработчика события нажатия кнопки `btnSort`.

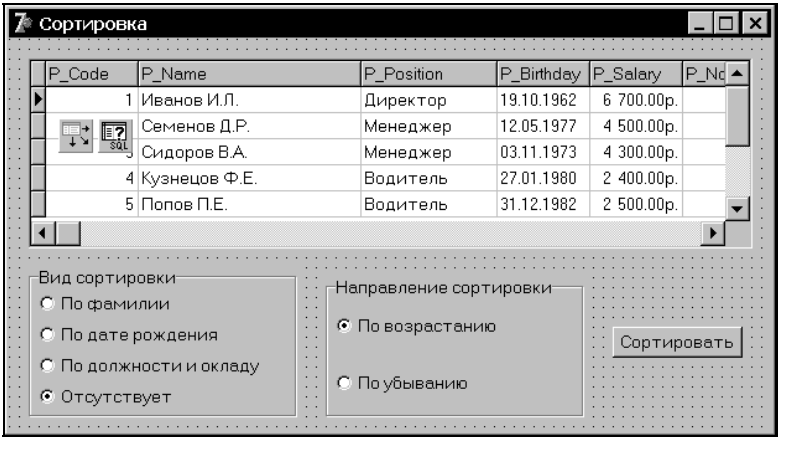


Рис. 20.1. Диалоговое окно сортировки набора данных

Листинг 20.1. Пример управления сортировкой

```
procedure TForm1.btnSortClick(Sender: TObject);
var s: string;
begin
  Query1.Close;
  Query1.SQL.Clear;
  Query1.SQL.Add('SELECT * FROM Personnel.db');
  case RadioGroup2.ItemIndex of
     0: s := '';
     1: s := 'DESC';
  end;
  case RadioGroup1.ItemIndex of
     0: s := 'ORDER BY P_Name ' + s;
     1: s := 'ORDER BY P_Birthday ' + s;
     2: s := 'ORDER BY P_Position ' + s + ', P_Salary ' + s;
     3: s := '';
  end;
  Query1.SQL.Add(s);
  Query1.Open;
end;
```

При нажатии кнопки `btnSort` для набора данных `Query1` происходят подготовка и выполнение SQL-запроса. Текст запроса формируется в зависимости от состояния переключателей, управляющих сортировкой.

Когда в SQL-запросе отсутствует параметр `ORDER`, по умолчанию записи упорядочиваются по первому полю. Поэтому в рассматриваемом примере отсутствие параметра сортировки и сортировка по полю `Name` приводят к одинаковому результату.

Соединение таблиц

В набор данных можно включать поля из разных таблиц, подобное включение называется *соединением (связыванием)* таблиц. Соединение таблиц может быть внутренним и внешним.

Внутреннее соединение представляет собой простейший случай, когда после слова `SELECT` перечисляются поля разных таблиц. Например:

```
SELECT P.Name, P.Position, P.Salary, A.Birthday, A.Note
      FROM Personnel.db P, Advanced.db A
```

Таблицы `Personnel` и `Advanced` содержат основные и дополнительные сведения о сотрудниках организации. Таблицы связаны отношением "один-к-одному", т. е. каждой записи первой таблицы соответствует одна запись второй таблицы. Результирующий набор данных является объединением полей двух таблиц так, будто дополнительные данные соединены с основными. В таблицах могут быть выбраны не все поля, что не меняет принципа соединения. Результирующий набор будет иметь вид:

```
Name          Position   Salary  Birthay    Note
Иванов Р.О.   Директор   6700    29.10.51
Петров А.П.   Менеджер   5200    3.4.62
```

```
Семенова И.И. Менеджер   5200    12.10.64
Кузнецов П.А. Секретарь  3600    7.11.81    Исп. срок до 31.6.2002
Васин Н.Е.    Водитель   2500    20.5.78
Попов А.Л.    Водитель   2400    3.2.75
```

Использование внутреннего соединения применимо не всегда. Например, при использовании внутреннего соединения таблиц с отношением "один-ко-многим" результат выполнения запроса может содержать избыточную информацию.

Рассмотрим в качестве примера запрос, в котором внутреннее соединение таблиц приводит к избыточной информации в результирующей выборке:

```
SELECT S.S_Name, C.C_Date, C.C_Move, S.S_Price
      FROM Store.db S, Cards.db C
```

Набор данных включает поля названия (`S_Name`) и цены (`C_Date`) товара из таблицы `Store` склада, а также поля (`C_Date`) даты и количества (`C_Move`) товара из таблицы `Cards` движения товара. Число записей набора данных является произведением числа записей в таблицах склада и движения товара. Результирующий набор данных может иметь вид:

```
S_Name       C_Date       C_Move   S_Price
Морковь      12.05.02     300      7,5
Морковь      12.05.02     200      7,5
Морковь      13.05.02     -7,50    7,5
Морковь      14.05.02     -30      7,5
Морковь      15.05.02     270      7,5
Морковь      17.05.02     200      7,5
Морковь      18.05.02     -120     7,5
Морковь      20.05.02     -10      7,5
Морковь      21.05.02     250      7,5
Яблоки       12.05.02     300      25
Яблоки       12.05.02     200      25
Яблоки       13.05.02     -7,50    25
...
Помидоры     20.05.02     -10      40
Помидоры     21.05.02     250      40
```

Результирующая выборка содержит избыточную информацию и большое число записей, что не помогает, а, наоборот, мешает пользователю. Поэтому при внутреннем соединении таблиц, связанных соотношением "один-ко-многим", обычно применяются критерии отбора, ограничивающие состав записей. Рассмотрим теперь такой запрос:

```
SELECT S.S_Name, C.C_Date, C.C_Move, S.S_Price
      FROM Store.db S, Cards.db C
      WHERE C.C_Code = S.S_Code
```

В отличие от предыдущего примера, число записей набора данных равно числу записей таблицы движения товара, т. к. отбираются записи, для которых совпадают значения полей кода. Результирующий набор данных будет таким:

```
S_Name       C_Date       C_Move   S_Price
Морковь      12.05.02     300      7,5
Яблоки       12.05.02     200      25
```

```
Морковь      13.05.02     -7,50     7,5
Морковь      14.05.02     -30       7,5
Морковь      15.05.02     270       7,5
Яблоки       17.05.02     200       25
Яблоки       18.05.02     -120      25
Морковь      20.05.02     -10       7,5
Помидоры     21.05.02     250       40
```

При внутреннем соединении все таблицы, поля которых указываются в SQL-запросе, являются равноправными.

При *внешнем* соединении таблиц можно указать, какая из таблиц будет *главной*, а какая — *подчиненной*. При использовании внешнего соединения операнд `FROM` имеет следующий формат:

```
FROM <Таблица1> [<Вид соединения>] JOIN <Таблица2>
     ON <Условия отбора>
```

Критерий отбора после слова `ON`, как и ранее, задает условие включения записей в набор данных, связываемые таблицы указываются слева и справа от слова `JOIN`. Какая из двух таблиц будет главной, определяет `Вид соединения`, который может иметь одно из следующих значений:

- `LEFT` (главная таблица указана слева);
- `RIGHT` (главная таблица указана справа) — по умолчанию.

Вот запрос, в котором используется внешнее связывание таблиц:

```
SELECT S.S_Name, C.C_Date, C.C_Move, S.S_Price
       FROM Store.db S LEFT JOIN Cards.db C
            ON C.C_Code = S.S_Code
```

Как и в предыдущем примере, связываются таблицы склада `Store` и движения товара `Cards`. Главной является таблица `Store`.

Модификация записей

Модификация записей заключается в редактировании записей, вставке в набор данных новых записей или удалении существующих записей. При реляционном доступе к БД операции модификации, как и другие операции, выполняются не над одиночными записями, а над группами записей. Группа может состоять и из одной записи.

Для модификации записей используются инструкции `UPDATE`, `INSERT` и `DELETE`, которые соответствующим образом изменяют записи и возвращают в качестве результата набор данных, состоящий из модифицированных записей, удовлетворяющих критерию отбора.

Редактирование записей

Редактирование записей представляет собой изменение значений полей в группе записей. Оно выполняется инструкцией `UPDATE` следующего формата:

```
UPDATE <Имя таблицы>
      SET <Имя поля1> = <Выражение1>,
      ...
      <Имя поляN> = <ВыражениеN>
      [WHERE <Условия отбора>];
```

После выполнения инструкции UPDATE для всех записей, удовлетворяющих условию отбора, изменяются значения полей. Имя поля указывает модифицируемое поле всей совокупности записей, а Выражение определяет значение, которое будет присвоено этому полю. Например:

```
UPDATE Personnel
      SET Salary = Salary + 200
      WHERE Salary < 1500
```

Если сотрудник имеет оклад менее 1500 (рублей), то оклад увеличивается на 200 (рублей).

Критерий отбора, указанный в операнде WHERE, не отличается от критерия, задаваемого в инструкции SELECT. Если он не задан, то изменяются значения всех указанных полей.

Вот соответствующий запрос:

```
UPDATE Store
      SET S_Price = S_Price * 1.28;
```

После его выполнения цена всех товаров увеличивается на 28%.

В одной инструкции UPDATE можно изменить значения *нескольких полей*, в этом случае для каждого из них указывается соответствующее значение. Например:

```
UPDATE Store
      SET S_Quantity = 0,
      S_Note = "Обнулено"
      WHERE S_Quantify BETWEEN — 0.5 AND 0.5
```

Для всех записей, у которых значение поля S_Quantity находится в диапазоне -0,5...0,5, этому полю присваивается значение 0, а в поле S_Note примечания записывается слово Обнулено.

Вставка записей

Вставка записей осуществляется с помощью инструкции INSERT, которая позволяет добавлять к таблицам одну или несколько записей. При добавлении одной записи инструкция INSERT имеет формат:

```
INSERT INTO <Имя таблицы>
      [(<Список полей>)]
      VALUES (<Список значений>);
```

В результате выполнения этой инструкции к таблице, имя которой указано после слова INTO, добавляется одна запись. Для добавленной записи заполняются поля, перечисленные в списке. Значения полей берутся из списка, расположенного после слова VALUES. Список полей и список значений должны соответствовать друг другу по числу

элементов и по типу. При присваивании значений для первого поля берется первое значение, для второго — второе и т. д. При этом порядок полей и значений может отличаться от порядка полей в таблице.

Пример запроса на добавление записи:

```
INSERT INTO Store
     (S_Name, S_Price, S_Quantity)
      VALUES ("Торшер", 499.9, 10);
```

Здесь в таблицу склада `Store` добавляется новая запись, в которой присваиваются значения полям названия товара, его цены и количества.

Список полей в инструкции `INSERT` может отсутствовать, в этом случае необходимо указать значения для всех полей таблицы. Порядок и тип этих значений должны соответствовать порядку и типу полей таблицы.

При добавлении к таблице сразу нескольких записей инструкция `INSERT` имеет формат:

```
INSERT INTO <Имя таблицы>
    (<Список полей>)
      Инструкция SELECT;
```

В данном случае значения полей новых записей определяются через значения полей записей, отобранных с помощью инструкции `SELECT`. Число добавленных записей равно числу отобранных записей. Список значений полей, возвращаемых инструкцией `SELECT`, должен соответствовать списку инструкции `INSERT` по числу и типу полей.

С помощью вставки группы записей можно скопировать данные из одной таблицы в другую, например, при резервном копировании или архивировании записей. При этом обе таблицы обычно имеют одинаковую структуру или их структуры частично совпадают.

Вот запрос на добавление нескольких записей:

```
INSERT INTO CardsArchives
       (Code, Move, Date)
       SELECT  C_Code, C_Move, C_Date
       FROM Cards
       WHERE C_Date BETWEEN 1.1.02 AND 31.12.02
```

В архивную таблицу `CardsArchives` добавляется группа записей из таблицы `Cards` движения товара. Для записей, сделанных в 2002 году, в архив копируются код товара, приход или расход и дата.

Если необходимо выполнить не копирование, а перемещение записей в архив, то после успешного копирования можно удалить записи в исходной таблице с помощью инструкции `DELETE`.

Замечание

Перемещение записей целесообразно оформлять в рамках транзакции, чтобы обеспечить надежность выполнения операции и сохранение целостности БД.

Удаление записей

Для удаления группы записей используется инструкция DELETE, имеющая формат:

```
DELETE FROM <Имя таблицы>
       [WHERE <Условия отбора>];
```

В результате выполнения этой инструкции из таблицы, имя которой указано после слова FROM, удаляются все записи, которые удовлетворяют критерию отбора.

Замечание

Если критерий отбора не задан, то из таблицы будут удалены все записи.

Вот соответствующий запрос:

```
DELETE FROM Store
       WHERE S_Quantity = 0
```

Из таблицы склада Store удаляются все записи о товаре, которого нет на складе. Отметим, что если с записями этой таблицы связаны записи другой таблицы, например, движения товара, то может потребоваться их предварительное удаление, что связано с действием бизнес-правил и налагаемыми ими ограничениями.

Статический и динамический запросы

Как отмечалось, в зависимости от способа формирования SQL-запрос может быть

- статическим;
- динамическим.

Текст *статического* SQL-запроса формируется при разработке приложения и в процессе выполнения приложения не может быть изменен. Такой запрос обычно используется в случаях, когда код запроса заранее известен и во время работы приложения не требует модификации.

Динамический SQL-запрос формируется или изменяется при выполнении приложения. Такой запрос обычно применяется, если его текст зависит от действий пользователя, например, при управлении сортировкой набора данных, когда в тексте запроса изменяются названия полей и добавляется или исключается описатель DESC.

Рассмотрим в качестве примера процедуру, в которой осуществляется формирование динамического запроса (листинг 21.2).

Листинг 21.2. Пример формирования динамического запроса

```
procedure TForm1.btnSortClick(Sender: TObject);
var str: string;
begin
  Query1.Close;
  Query1.SQL.Clear;
  Query1.SQL.Add('SELECT * FROM Personnel.db');
```

```
  case RadioGroup1.ItemIndex of
     0: str := 'ORDER BY Name ';
     1: str := 'ORDER BY BirthDay ';
  end;
  case RadioGroup2.ItemIndex of
     0:;
     1: str := str + ' DESC';
  end;
  Query1.SQL.Add(str);
  Query1.Open;
end;
```

При нажатии кнопки `btnSort` список сотрудников сортируется по полям `Name` или `BirthDay`. Пользователь управляет выбором поля с помощью группы переключателей `RadioGroup1`. В группе `RadioGroup2` выбирается порядок сортировки.

Для настройки статического SQL-запроса во время выполнения приложения в его тексте можно использовать параметры. *Параметр* — это специальная переменная, перед именем которой ставится двоеточие в тексте запроса. Двоеточие не является частью имени параметра и ставится только в тексте запроса. Как и для обычных переменных программы, в процессе выполнения приложения вместо параметра подставляется его значение.

Параметры удобно использовать для передачи в текст SQL-запроса внешних значений. Например, если необходимо вывести фамилии сотрудников с окладом, не менее указанного в редакторе `Edit1`, то организовать формирование и выполнение динамического запроса можно так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.SQL.Clear;
  Query1.SQL.Add('SELECT Name, Position, Salary');
  Query1.SQL.Add('FROM Personnel.db');
  Query1.SQL.Add('WHERE Salary >= ' + Edit1.Text);
  Query1.Open;
end;
```

С помощью параметров эту задачу можно решить проще, например, через включение в текст запроса параметра `prmSalary`:

```
SELECT Name, Position, Salary
  FROM Personnel.db
  WHERE Salary >= :prmSalary
```

Система Delphi автоматически учитывает все указанные в SQL-запросе параметры в специальном списке параметров, являющемся для набора данных `Query` значением свойства `Params` типа `TParams`, представляющим собой массив. Это свойство позволяет получить доступ к каждому параметру как при разработке, так и при выполнении приложения. Чтобы обратиться к параметру во время выполнения приложения, следует указать его номер (индекс) в списке параметров. Например, для обращения ко второму параметру указывается `Params[1]`. На этапе разработки приложения можно вызвать Ре-

дактор параметров (рис. 20.2) в Инспекторе объектов. Тип каждого параметра необходимо указать в свойстве `DataType` типа `TFieldType`. Например, для параметра `prmSalary` установлен тип `ftFloat`.

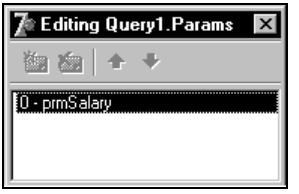


Рис. 20.2. Окно Редактора параметров

В последующем перед выполнением запроса вместо параметра необходимо подставить его значение, в данном случае из редактора `Edit1`. Далее приведен код обработчика события нажатия кнопки `Button1`, выполняющий указанное действие.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.ParamByName('prmSalary').AsFloat := StrToFloat(Edit1.Text);
  Query1.Open;
end;
```

Для доступа к параметру во время выполнения приложения используется метод `ParamByName`, отличающийся от аналогичного метода `FieldByName` тем, что вместо имени поля указывается имя параметра.

При использовании параметров можно не изменять текст SQL-запроса во время выполнения приложения и, тем не менее, передавать в него различные значения. То есть статический запрос как бы превращается в динамический.

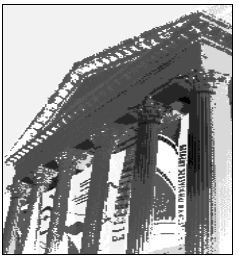
Замечание

Статические запросы, в которых использованы параметры, иногда также называют *изменяющимися* (т. е. фактически динамическими).

Обычно текст SQL-запроса проверяется и выполняется при каждом открытии набора данных `Query`. Если текст запроса при выполнении приложения не изменяется, то его можно предварительно подготовить, а после этого только использовать такой подготовленный к выполнению запрос. Это позволяет ускорить обработку статических запросов, в том числе имеющих параметры.

Подготовку запроса к выполнению осуществляет метод `Prepare`, который можно вызывать при создании формы. Чтобы определить, была ли произведена предварительная подготовка запроса, необходимо проанализировать свойство `Prepared` типа `Boolean`, которое после вызова метода `Prepare` устанавливается в значение `True`.

Если текст подготовленного к выполнению запроса изменился (к изменению значений параметров это не относится), то автоматически вызывается метод `UnPrepare`, и свойство `Prepared` устанавливается в `False`.



ГЛАВА 21

Технология dbExpress

Общая характеристика

В основе технологии dbExpress лежит использование множества легковесных драйверов, компонентов, объединяющих соединения, транзакции, запросы и наборы данных, а также интерфейсов, реализующих универсальный доступ к соответствующим функциям.

По сравнению с использованием механизма BDE технология dbExpress обеспечивает построение более легковесных (по объему кода) приложений для работы с базами данных. При ее применении для доступа к данным используются SQL-запросы. Технология dbExpress обеспечивает легкую переносимость приложений, допускает работу приложений баз данных под управлением Windows и Linux.

Для использования технологии dbExpress достаточно включить в распространяемое приложение динамически подключаемую библиотеку с драйвером, взаимодействующим с клиентским программным обеспечением для нужного сервера базы данных (рис. 21.1).

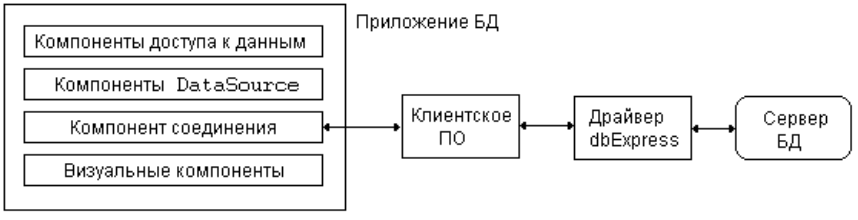


Рис. 21.1. Схема доступа к данным с помощью dbExpress

Драйверы, используемые для доступа к различным серверам баз данных по технологии dbExpress, реализованы в виде динамически подключаемых библиотек (DLL) (табл. 21.1). Драйверы dbExpress в составе Delphi размещаются в папке ...\Borland \Delphi7\BIN.

Таблица 21.1. *Драйверы БД*

Драйвер сервера БД	Динамическая библиотека
Interbase dbExpress	Dbexpint.dll
MySQL dbExpress	Dbexpmysql.dll
Old MySQL dbExpress	Dbexpmys.dll
DB2 dbExpress	Dbexpdb2.dll
Oracle dbExpress	Dbexpora.dll
dbExpress for Microsoft SQL Server	Dbexpmss.dll
Informix dbExpress	Dbexpinf.dll

На странице **dbExpress** Палитры компонентов Delphi находятся компоненты, используемые в технологии dbExpress: `SQLConnection` (`Database`), `SQLDataSet`, `SQLQuery` (`Query`), `SQLStoredProc` (`StoredProc`), `SQLTable` (`Table`), `SQLMonitor` (утилита SQL Monitor) и `SimpleDataSet` (`BDEClientDataSet`) (рис. 21.2).

Рис. 21.2. Страница **dbExpress**

Для наглядности в круглых скобках указаны аналоги этих компонентов, используемые в случае механизма BDE. Как видим из приведенного списка, у компонента `SQLDataSet` нет аналога для механизма BDE. Кроме того, ряд компонентов из механизма BDE не имеет аналогов для технологии dbExpress.

Определенным недостатком технологии dbExpress является то, что несколько из перечисленных компонентов (`SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`) являются однонаправленными наборами данных, в которых отсутствует буферизация. Эти наборы данных обеспечивают более быстрый доступ к данным и предъявляют меньшие требования к ресурсам, но при этом на них накладываются заметные ограничения. Для компонента `SimpleDataSet` большинство из этих ограничений не действует.

Установление соединения с сервером

Для установления соединения с сервером базы данных служит компонент `SQLConnection`, который представляет собой аналог компонента `DataBase` в BDE.

Этот компонент взаимодействует с двумя файлами, расположенными в каталоге ...\Common Files\Borland Shared\DBExpress. Файл dbxdrivers.ini содержит список инсталлированных драйверов серверов БД и для каждого драйвера список динамически подключаемых библиотек и параметров соединений, установленных по умолчанию. Список соединений с параметрами соединений содержится в файле dbxconnections.ini.

Поместив компонент `SQLConnection` в форму (или модуль данных) на этапе разработки приложения, можно выбрать одно из существующих соединений, либо создать новое соединение с помощью диалогового окна Редактора соединений (рис. 21.3). Вызов указанного окна можно выполнить выбором пункта **Edit Connection Properties** контекстного меню компонента.

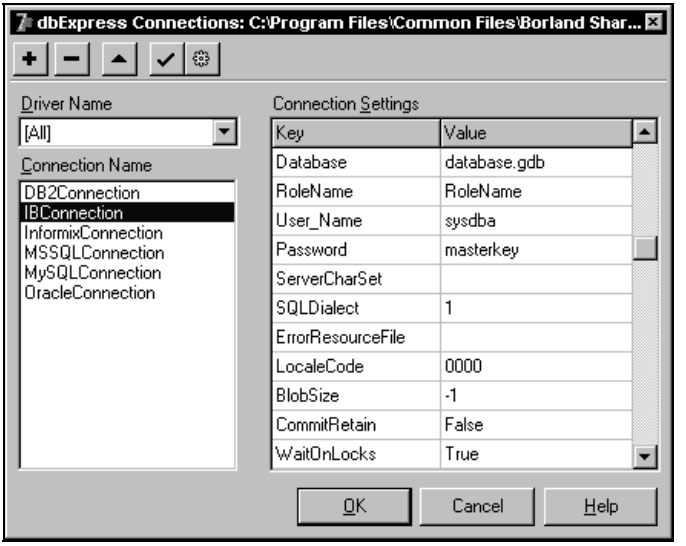


Рис. 21.3. Диалоговое окно Редактора соединений

Редактор соединений позволяет настроить существующее соединение dbExpress или создать новое соединение, а также проверить правильность настроек с помощью кнопки ☑ (**Test Connection**). Параметры соединения можно настраивать также с помощью Инспектора объектов (свойство `Params` типа `TStrings`).

Параметры соединения для основных серверов баз данных приведены в табл. 21.2.

Таблица 21.2. *Параметры соединений для серверов БД*

Сервер БД	Параметр	Значение
Все серверы	`BlobSize`	Ограничение на объем пакета для данных типа BLOB
	`DriverName`	Имя драйвера
	`ErrorResourceFile`	Файл сообщений об ошибках
	`LocaleCode`	Код локализации, определяющий влияние национальных символов на сортировку данных
	`User_Name`	Имя пользователя
	`Password`	Пароль
	`<имя сервера БД>TransIsolation`	Уровень изоляции транзакций для сервера с заданным именем

Таблица 21.2 *(окончание)*

Сервер БД	Параметр	Значение
InterBase	CommitRetain	Поведение курсора по завершению транзакции: True — обновление; False — удаление
	RoleName	Роль пользователя
	SQLDialect	Используемый диалект SQL
	Trim Char	Возможность удаления из строковых значений полей пробелов, дополняющих их до полной строки
	WaitOnLocks	Разрешение на ожидание занятых ресурсов
	DataBase	Имя файла базы данных (с расширением gdb)
DB2	DataBase	Имя клиентского ядра
MySQL	DataBase	Имя базы данных
Microsoft SQL Server 2000, MySQL, Informix	HostName	Имя компьютера, на котором установлен сервер
Microsoft SQL Server 2000, Informix	DataBase	Псевдоним (алиас) имени базы данных
Microsoft SQL Server 2000, Oracle	OS Autenification	Использование учетной записи текущего пользователя операционной системы при доступе к ресурсам сервера
Informix, Oracle	Trim Char	Возможность удаления из строковых значений полей пробелов, дополняющих значение до полной строки
Oracle	AutoCommit	Флаг завершения транзакции. Устанавливается сервером
	BlockingMode	Режим завершения запроса. True — соединение дожидается окончания запроса; False — начинает выполнение следующего запроса
	Multiple Transaction	Возможность управления несколькими транзакциями в одном сеансе
	DataBase	Запись базы данных в файле TNSNames.ora

После описанной настройки параметров соединений для компонента SQLConnection с помощью Инспектора объектов можно выбрать соединение (свойство ConnectionName) для нужного сервера баз данных. При этом автоматически устанавливаются значения связанных с ним свойств: DriverName (имя драйвера); LibraryName (имя динамически подключаемой библиотеки драйвера); Params (параметры соединения) и VendorLib (имя динамически подключаемой библиотеки клиентской части сервера).

Открытие соединения с сервером БД осуществляется заданием свойству Connected типа boolean значения True. Например, при выполнении приложения это можно сделать с помощью кода:

```
SQLConnection1.Connected := True;
```

Соответственно закрыть соединение с сервером можно путем задания этому свойству значения `False`. Названные операции могут быть выполнены также с помощью методов:

```
procedure Open;
procedure Close;
```

С помощью свойства `LoginPrompt` типа `boolean` выполняется задание необходимости (`True`) или ненужности (`False`) отображения окна авторизации для ввода имени пользователя (`User_Name`) и пароля (`Password`) при каждой попытке соединения с сервером. В последнем случае эти значения будут браться из файла dbxconnections.ini.

После открытия соединения все компоненты dbExpress, инкапсулирующие наборы данных и связанные с открытым компонентом `SQLConnection`, получают доступ к базе данных.

При открытии и закрытии соединения можно воспользоваться любым из доступных событий, например:

```
property AfterConnect: TNotifyEvent;
property AfterDisconnect: TNotifyEvent;
property BeforeConnect: TNotifyEvent;
property BeforeDisconnect: TNotifyEvent;
property OnLogin: TSQLConnectionLoginEvent;
```

Последнее событие наступает в случае, если свойству `LoginPrompt` установлено значение `True`. Если это свойство имеет значение `True`, и нет обработки события `OnLogin`, то пользователю будет предложен стандартный диалог авторизации.

Текущее состояние соединения указывает `ConnectionState` типа `TConnectionState`, тип которого описан так:

```
TConnectionState = (csStateClosed, csStateOpen, csStateConnecting,
          csStateExecuting, csStateFetching, csStateDisconnecting);
```

Указанные в определении типа значения соответствуют следующим состояниям соединения: закрыто, открыто, установление соединения, ожидание исполнения переданного SQL-запроса, получение данных с сервера, завершение соединения.

Компонент `SQLConnection` позволяет выполнять SQL-запросы с помощью следующих функций.

Функция `Execute(const SQL: string; Params: TParams; ResultSet: Pointer = Nil): Integer` выполняет запрос, определенный значением константы `SQL` с параметрами `Params`, используемыми в SQL-запросе. Используемые в запросе параметры должны иметь тот же порядок следования, что и в `Params`, и в инструкции SQL. Если SQL-запрос не содержит параметров, то свойству `Params` требуется задать значение `Nil`. Если при выполнении SQL-запроса возвращается результат, то в параметр `ResultSet` помещается указатель на объект типа `TCustomSQLDataSet`, содержащий результат.

При отсутствии в запросе параметров и возвращаемых записей рекомендуется использовать функцию `ExecuteDirect(const SQL: string): Integer`, которая возвращает значение 0 при успешном завершении запроса или код ошибки в противном случае.

Подобно своим аналогам в BDE *(см. главу 19)* компонент `SQLConnection` позволяет выполнять запуск, фиксацию и откат *транзакций* соответственно с помощью методов `StartTransaction`, `Commit` и `Rollback`.

Компоненты доступа к данным

Как отмечалось, ряд компонентов доступа к данным по технологии dbExpress (`SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`) принадлежит к однонаправленным наборам данных, в которых отсутствует буферизация, и при этом на них накладываются заметные ограничения.

В однонаправленных наборах данных используются однонаправленные курсоры. С их помощью допускается только получать данные, из методов навигации по набору данных поддерживаются лишь методы `First` и `Next`, подразумевающие последовательный перебор записей от начала к концу.

При использовании однонаправленных наборов данных отсутствует возможность прямого редактирования данных, т. к. для этого нужно размещение в буфере результатов редактирования. При этом соответствующее свойство названных компонентов (`CanModify`) всегда имеет значение `False`. Для однонаправленных наборов данных возможности редактирования данных все же доступны, например, путем задания оператора `UPDATE` языка SQL. Кроме того, для однонаправленных наборов данных по тем же причинам имеют место ограничения по выполнению фильтрации. Это ограничение также можно обойти путем указания параметров фильтрации в SQL-запросах.

Имеются также ограничения по отображению данных с помощью компонентов страницы **Data Controls**. В частности, нельзя использовать компоненты `DBGrid` и `DBCtrlGrid`, а также компоненты синхронного просмотра. Для компонента навигатора требуется отключить кнопки возврата на последнюю запись и перехода на последнюю запись. Остальные компоненты можно использовать как обычно.

Для компонента `SimpleDataSet` большинство из указанных ограничений не действует. Он использует внутренние компоненты типа `TSQLDataSet` и `TDataSetProvider` для получения и изменения данных.

Перед использованием любой из указанных компонентов доступа к данным следует подключить к серверу БД. Для этого достаточно его свойству `SQLConnection` в качестве значения установить имя компонента соединения `SQLConection`.

Компоненты доступа к данным можно использовать, поместив в форме, либо динамически — создав при выполнении приложения.

Универсальный доступ к данным

Для обеспечения универсального однонаправленного доступа к данным БД по технологии dbExpress служит компонент `SQLDataSet`. Он позволяет получать все записи из таблицы БД, производить выборку данных путем выполнения SQL-запросов или выполнять хранимые процедуры.

Для определения типа выполняемых действий (с запросами, таблицами или хранимыми процедурами) свойству `CommandType` рассматриваемого компонента нужно установить соответственно одно из трех возможных значений:

- `ctQuery` — в свойстве `CommandText` указывается SQL-запрос;
- `ctTable` — в свойстве `CommandText` указывается имя таблицы на сервере БД, при этом компонент автоматически генерирует SQL-запрос для получения всех записей для всех полей таблицы;
- `ctStoredProc` — в свойстве `CommandText` указывается имя хранимой процедуры.

В случае выбора значения `ctQuery` текст SQL-запроса можно ввести в поле свойства `CommandText` с помощью Инспектора объектов или редактора построения запроса `CommandTextEditor` (рис. 21.4). В списке **Tables** доступны имена таблиц в базе данных, а в списке **Fields** — имена полей выбранной таблицы.

С помощью кнопок **Add Table to SQL** и **Add Field to SQL** можно добавить нужные таблицы и поля. При этом в поле **SQL** отображается автоматически сформированный SQL-запрос. После нажатия кнопки **OK** в окне редактора мы получаем нужное значение свойства `CommandText`.

При выполнении приложения аналогичные действия (как на рис. 21.4) для выполнения SQL-запроса можно задать так:

```
SQLDataSet1.CommandType := ctQuery;
SQLDataSet1.CommandText := 'select COUNTRY, CURRENCY from DEPARTMENT, DEPT_NO,
DEPARTMENT from COUNTRY ';
SQLDataSet1.ExecSQL;
```

При выборе значения `ctTable` в свойстве `CommandText` указывается просто имя таблицы, например, так:

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'COUNTRY';
SQLDataSet1.ExecSQL;
```

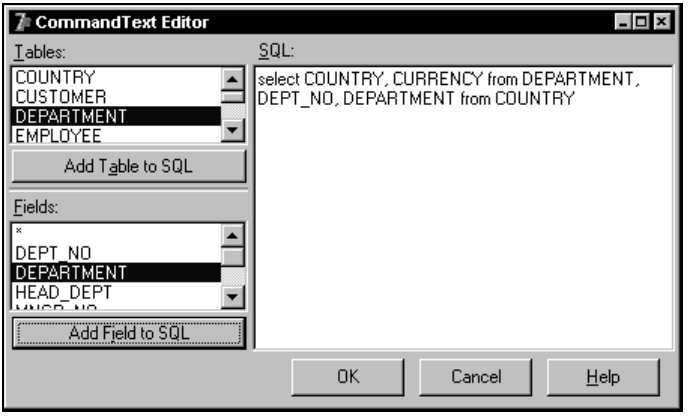


Рис. 21.4. Редактор построения SQL-запроса

Для открытия набора данных нужно установить значение `True` свойству `Active` или использовать метод `Open`. Если же SQL-запрос или хранимая процедура не возвращают набор данных, для их выполнения используется метод

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Параметр `ExecDirect` определяет необходимость подготовки параметров перед выполнением инструкции. Если параметры запроса или процедуры существуют, параметр `ExecDirect` должен иметь значение `False`.

При необходимости сортировки данных используют инструкцию `ORDER BY` в случае варианта `ctQuery`, либо устанавливают с помощью редактора SortFields Editor значение свойства `SortFieldNames` в случае варианта `ctTable`. Для варианта `ctStoredProc` порядок сортировки определяется в хранимой процедуре.

При необходимости в SQL-запросе или в хранимой процедуре можно использовать параметры, позволяющие настраивать запрос или процедуру, не изменяя их код. Параметры задаются с помощью свойства `Params`, которое является коллекцией объектов типа `TParam`.

Свойство `DataSource` типа `TDataSource` компонента `SQLDataSet` позволяет связать два набора данных по схеме "мастер — детальный" *(см. главу 14)*.

Метод `SetSchemaInfo` компонента `SQLDataSet` позволяет получить метаданные (список таблиц на сервере, список системных таблиц, информацию о хранимых процедурах, информацию о полях таблицы, параметры хранимой процедуры).

Рассмотрим пример формы (рис. 21.5) приложения БД, в котором с помощью компонентов `SQLDataSet` выполняется доступ к данным по технологии dbExpress. Пусть требуется обеспечить возможность просмотра и редактирования полей данных `'CUST_NO'`, `'CUSTOMER'` и `'PHONE_NO'` таблицы `CUSTOMER`, принадлежащей базе данных EMPLOYEE.GDB сервера InterBase.

Код модуля формы для решения поставленной задачи приведен в листинге 21.1.

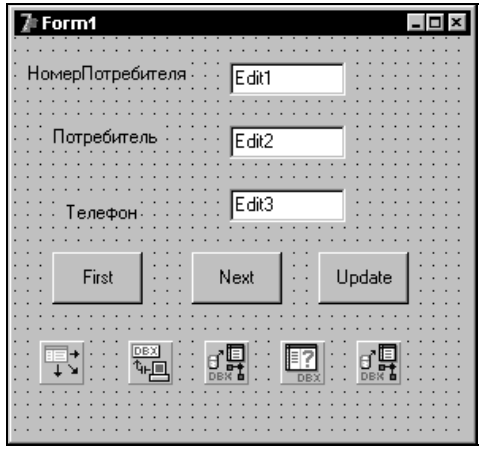


Рис. 21.5. Вид формы при разработке

Листинг 21.1. Приложение для доступа к данным по технологии dbExpress

```
unit UnitSQLDataSet;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, DBXpress, FMTBcd, DB, SqlExpr, StdCtrls, Mask, DBCtrls;

type
  TForm1 = class(TForm)
    SQLConnection1: TSQLConnection;
    SQLDataSet1:    TSQLDataSet;
    DataSource1:    TDataSource;
    Label1:         TLabel;
    Label2:         TLabel;
    Button1:        TButton;
    Button2:        TButton;
    Button3:        TButton;
    SQLQuery1:      TSQLQuery;
    SQLDataSet2:    TSQLDataSet;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    procedure SQLDataSet1AfterScroll(DataSet: TDataSet);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
implementation

{$R *.dfm}

procedure TForm1.SQLDataSet1AfterScroll(DataSet: TDataSet);
begin
  Edit1.Text:=SQLDataSet1.FieldByName('CUST_NO').AsString;
  Edit2.Text:=SQLDataSet1.FieldByName('CUSTOMER').AsString;
  Edit3.Text:=SQLDataSet1.FieldByName('PHONE_NO').AsString;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  SQLDataSet1.First;
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  SQLDataSet1.Next;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  with SQLDataSet2 do
    try
      CommandText := 'UPDATE CUSTOMER SET CUST_NO = :Cust_no, CUSTOMER =
:Customer, PHONE_NO = :Phone_no WHERE CUST_NO = :CUST_NO';
      Params[0].AsString := Edit1.Text;
      Params[1].AsString := Edit2.Text;
      Params[2].AsString := Edit3.Text;
      ExecSQL;
    except
      MessageDlg('Customer info fixed error',mtError,[mbOK],0);
    end;
end;
end.
```

Для просмотра и редактирования трех указанных полей таблицы `CUSTOMER` используются два компонента — `SQLDataSet1` и `SQLDataSet2` соответственно. Оба компонента подсоединены к таблице сервера InterBase через компонент `SQLConnection1`. Первый компонент `SQLDataSet1` служит для просмотра записей таблицы `CUSTOMER`. Второй компонент `SQLDataSet2` применяется для фиксации внесенных изменений (три компонента типа `TEdit`) в текущую запись в таблице на сервере БД.

Для заполнения компонентов типа `TEdit` при навигации по набору данных используется обработчик события `AfteScroll` для компонента `SQLDataSet1`. Навигация по набору данных выполняется с помощью обработчиков событий `OnClick` для кнопок **First** и **Next**. Фиксация внесенных в просматриваемую запись изменений в таблице на сервере осуществляется с помощью SQL-запроса `UPDATE`, размещенного в обработчике события `OnClick` для кнопки **Update**. В параметрах запроса передаются текущие значения полей из компонентов `Edit1`, `Edit2` и `Edit3` (рис. 21.6).

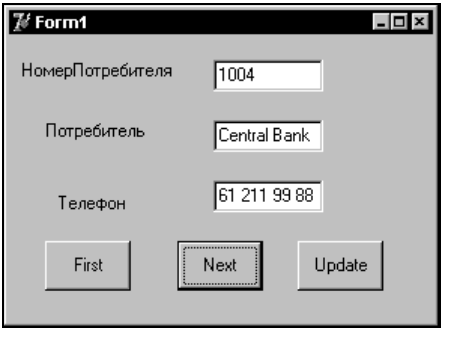


Рис. 21.6. Вид формы при выполнении приложения

Отметим, что необходимость использования компонента SQLDataSet2 для фиксации внесенных изменений в таблице на сервере вызвано тем, что компонент SQLDataSet1 является однонаправленным набором данных. Заметим также, что в нашем примере нельзя вместо компонентов Edit использовать компоненты DBEdit, т. к. здесь они не позволяют выполнять редактирование содержимого полей.

Просмотр таблиц

Для просмотра таблиц по технологии dbExpress служит компонент SQLTable. Компонент SQLTable генерирует SQL-запрос для получения всех строк и полей указанной таблицы.

Имя таблицы определяется с помощью свойства TableName, и при подключении компонента к соединению его можно выбрать в комбинированном списке в окне Инспектора объектов.

Для получения табличного набора данных компонент SQLTable с помощью метода

```
procedure PrepareStatement; override;
```

генерирует запрос на сервер БД.

Порядок следования данных в наборе определяется свойством IndexFieldNames типа String или IndexName типа String.

Список индексов таблицы можно получить с помощью метода GetIndexNames(List: TStrings) в качестве значения параметра List.

Связь между двумя наборами данных "главный — подчиненный" устанавливается с помощью свойств MasterFields и MasterSource. На этапе разработки приложения двойным щелчком на свойстве MasterFields в окне Инспектора объектов можно открыть диалоговое окно редактора связей для визуального построения отношения "мастер — детальный".

Компонент SQLTable является однонаправленным курсором, тем не менее, он допускает удаление записей из таблицы на сервер БД с помощью метода DeleteRecords.

Выполнение SQL-запроса

Для выполнения SQL-запроса на сервере БД служит компонент SQLQuery. Он позволяет представлять результаты выполнения запросов с помощью инструкции SELECT или осуществлять действия по изменению БД путем выполнения инструкций INSERT, DELETE, UPDATE, ALTER TABLE и т. п.

Для компонента SQLQuery текст SQL-запроса определяется как значение свойства SQL типа TStrings. На этапе разработки приложения текст SQL-запроса может быть набран непосредственно в окне редактора строк (или редактора кода), открываемого двойным щелчком на свойстве SQL в окне Инспектора объектов.

При выполнении приложения очистка содержимого свойства SQL компонента SQLQuery и его модификация могут быть выполнены, к примеру, с помощью следующего кода:

```
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.Add('SELECT ' + Edit1.Text + ' FROM ' + Edit2.Text);
```

```
if Length(Edit3.Text) <> 0 then
  SQLQuery1.SQL.Add('ORDER BY ' + Edit3.Text)
```

Свойство `Text` типа `String` в качестве значения содержит строковое представление SQL-запроса в виде одной строки, а не нескольких, как это может быть в случае свойства `SQL`.

Свойство `Active` типа `Boolean` определяет, открыт набор данных или нет. Это свойство обычно используют, чтобы определить или установить заполнение набора данными.

Открыть набор данных SQL-запроса также можно либо с помощью метода `Open`, либо с помощью метода

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Здесь значение `False` параметра `ExecDirect` означает, что запрос не имеет настраиваемых параметров.

Свойство `DataSource` связывает набор данных SQL-запроса с другим (главным) набором данных. Этот набор данных служит для получения параметров запроса в случае, когда предусматривается параметризованный запрос, но приложение этими параметрами не обеспечивает.

Выполнение хранимых процедур

Для выполнения хранимых процедур, размещенных на сервере БД, служит компонент `SQLStoredProc`.

Имя хранимой процедуры задает свойство `StoredProcName` типа `String`. Для задания параметров хранимой процедуры предназначено свойство `Params` типа `TParams`. При обращении к параметрам хранимой процедуры целесообразно использовать метод `ParamByName`. Это обусловлено тем, что при работе с некоторыми серверами порядок следования параметров до и после выполнения процедуры может меняться.

Для подготовки хранимой процедуры к выполнению на сервере служит метод `PrepareStatement(var RecordsAffected: Integer): TCustomSQLDataSet`. При его вызове сервером БД выделяются ресурсы и связываются их параметры. Поименованные параметры временно преобразуются к непоименованным параметрам, поскольку dbExpress поименованные параметры не поддерживает.

Если хранимая процедура не возвращает набор данных, то ее запускают с помощью метода

```
function ExecProc: Integer; virtual;
```

В противном случае используется метод `Open` либо свойству `Active` задают значение `True`.

Компонент редактирования набора данных

Для редактирования набора данных (получения данных, их кэширования и отправления измененных данных на сервер) предназначен компонент `SimpleDataSet`. Этот компонент использует двунаправленный курсор и позволяет редактировать данные, но в

режиме редактирования. Тем самым он исправляет основные недостатки рассматриваемой технологии.

Для подготовки компонента `SimpleDataSet` к работе с данными нужно с помощью свойства `Connection` связать его с компонентом соединения `SQLConnection`. В качестве альтернативы можно с помощью подсвойства `ConnectionName` свойства `Connection` задать тип соединения непосредственно.

Произведенные над данными изменения размещаются в локальном кэше, в связи с этим для подтверждения изменений и отправки данных на сервер БД используют метод

```
function ApplyUpdates(MaxErrors: Integer); Integer; virtual;
```

Здесь параметр метода определяет число ошибок, допустимых при передаче данных.

Локальный кэш компонента после сохранения изменений на сервере можно очистить от данных с помощью метода

```
function Reconcile(const Results: OleVariant): Boolean;
```

Отмена локальных изменений данных может быть выполнена с помощью метода

```
procedure CancelUpdates;
```

Пересылка данных между сервером и рассматриваемым компонентом осуществляется с помощью пакетов. Размер пакета (по числу записей) можно задать с помощью свойства `PacketRecords` типа `Integer`. По умолчанию устанавливается значение –1, которое означает, что один пакет должен содержать все записи набора данных.

Если значение свойства `PacketRecords` больше 0, то оно определяет число записей, которые можно получить в пакете от провайдера с помощью метода

```
function GetNextPacket: Integer;
```

Если значение свойства `PacketRecords` равно 0, то в пакете передаются только метаданные.

Свойство `DataSize` типа `Integer` устанавливает размер (в байтах) текущего пакета, доступного с помощью свойства `Data` типа `OleVariant`.

На общее число записей в источнике данных указывает свойство `RecordCount` типа `Integer`, номер текущей записи определяет свойство `RecNo` типа `Integer`.

Изменения в текущей записи можно отменить с помощью метода

```
procedure RevertRecord;
```

Отменить последнюю операцию по изменению клиентского источника данных можно с помощью метода

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

Здесь значение параметра определяет, где будет установлен курсор после восстановления записи: `True` — на восстановленной записи, `False` — на текущей записи.

Обновить значение полей для текущей записи с сервера можно с помощью метода `RefreshRecord`.

Рассмотрим пример формы (рис. 21.7) приложения БД, в котором с помощью компонента `SimpleDataSet1` выполняется доступ к данным по технологии dbExpress. Пусть требуется обеспечить возможность просмотра и редактирования записей таблицы `DEPARTMENT`, принадлежащей базе данных EMPLOYEE.GDB сервера InterBase.

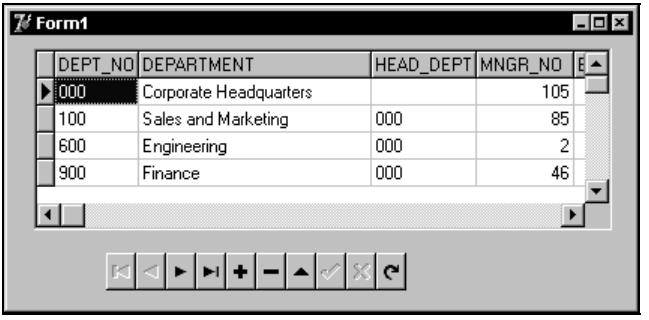


Рис. 21.7. Вид формы `UnitSimpleDataSet` при выполнении приложения

Код модуля формы для решения поставленной задачи приведен в листинге 21.2.

Листинг 21.2. Пример использования компонента `SimpleDataSet1`

```
unit UnitSimpleDataSet;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, FMTBcd, DBXpress, DB, SqlExpr, ExtCtrls, DBCtrls,
  Grids, DBGrids, DBClient, SimpleDS;
type
  TForm1 = class(TForm)
    SimpleDataSet1: TSimpleDataSet;
    SQLConnection1: TSQLConnection;
    DBGrid1:        TDBGrid;
    DataSource1:    TDataSource;
    DBNavigator1:   TDBNavigator;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure SimpleDataSet1AfterPost(DataSet: TDataSet);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  SimpleDataSet1.Open;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  SimpleDataSet1.Active:=False;
end;

procedure TForm1.SimpleDataSet1AfterPost(DataSet: TDataSet);
begin
  SimpleDataSet1.ApplyUpdates(-1);
end;
end.
```

Компонент `SimpleDataSet1` работает в табличном режиме. При разработке формы подсвойству `CommandType` свойства `DataSet` задано значение `ctTable`, с помощью подсвойства `CommandText` указано имя таблицы `DEPARTMENT`.

Для отображения записей редактируемого набора данных и навигации по нему используются компоненты `DBGrid1` и `DBNavigator1` соответственно. Фиксация внесенных при редактировании изменений и отправка данных на сервер осуществляются в обработчике события `AfterPost`, возникающего при нажатии кнопки `Post` (утвердить результат изменения записи) компонента `DBNavigator1` и при переходе в компоненте `DBGrid1` на другую строку *(см. главу 18)*.

Отладка соединения с сервером

Для получения информации о местах ошибок в SQL-запросах при отладке соединения приложения с сервером БД служит компонент `SQLMonitor`. Он осуществляет перехват сообщений между соединением и сервером баз данных и помещает их в список строк, определяемый свойством `TraceList` типа `TStrings`.

Для начала работы с компонентом нужно установить связь с соединением через свойство `SQLConnection` и активизировать его, задав свойству `Active` значение `True`.

Перед каждой записью сообщений в список строк возникает событие `OnTrace` типа

```
TTraceEvent = procedure(Sender: TObject; CBInfo: pSQLTRACEDesc;
                        var LogTrace: Boolean) of object;
```

а сразу после записи в список возникает событие `OnLogTrace` типа

```
TTraceLogEvent = procedure (Sender: TObject; CBInfo: pSQLTRACEDesc)    of object;
```

Содержимое списка можно сохранить в файле на жестком диске с помощью метода `SaveToFile`.

При задании свойству `AutoSave` типа `Boolean` значения `True` данные о проходящих командах автоматически заносятся в текстовый файл с именем, заданным значением свойства `FileName` типа `String`.

Свойство `MaxTraceCount` типа `Integer` определяет максимальное число контролируемых команд, а также управляет процессом контроля. При значении –1 ограничения снимаются, а при значении 0 контроль не выполняется.

На число сохраненных в списке команд указывает свойство `TraceCount` типа `Integer`.

Рассмотрим следующий пример: пусть требуется выполнять оперативное отображение информации о последнем сообщении в окне многострочного редактора (компонент `memo1`).

Для решения названной задачи подходит следующий код:

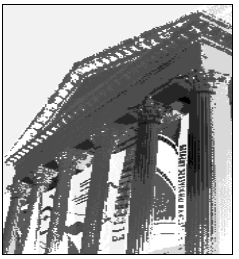
```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CBInfo: pSQLTRACEDesc);
var LogFileName: string;
begin
   with Sender as TSQLMonitor do
   begin
     memo1.Lines.Clear;
     memo1.Lines:=TraceList;
     TraceList.Clear; {очистка списка сообщений}
   end;
end;
```

Как видно из приведенного кода, мы использовали обработчик события `OnLogin`.

Замечание

Для автономного dbExpress-приложения нужны две динамически подключаемые библиотеки (DLL). Первая библиотека содержит драйвер dbExpress, например, для сервера БД InterBase это dbexpint.dll (см. табл. 21.1). Вторая библиотека есть midas.dll, используемая для поддержки компонента `SimpleDataSet`. Обе библиотеки могут быть откомпилированы вместе с проектом и включены в exe-файл распространяемого приложения.

При использовании компонента `SimpleDataSet` в приложении Delphi может возникать ошибка, сопровождаемая сообщением: "Error loading Midas.dll". Возможной причиной ее появления является неправильная регистрация названной библиотеки в среде операционной системы. Для выполнения регистрации библиотеки Midas.dll с целью устранения этой ошибки при работе под управлением Windows 2000 достаточно запустить на выполнение утилиту regsvr32.exe. Обычно она размещается в системном каталоге Windows, например, \winnt\system32.



ГЛАВА 22

Технология ADO

Общая характеристика

Технология Microsoft ActiveX Data Objects (ADO) представляет собой универсальный механизм доступа к различным источникам данных из приложений баз данных. Основу технологии ADO составляет использование набора интерфейсов общей модели объектов COM, описанных в спецификации OLE DB. Достоинством этой технологии является то, что базовый набор интерфейсов OLE DB имеется в каждой современной операционной системе Microsoft. Отсюда следует простота обеспечения доступа приложения к данным. При применении технологии ADO (рис. 22.1) приложение БД может использовать данные из электронных таблиц, таблиц локальных и серверных баз данных, XML-файлов и т. д.

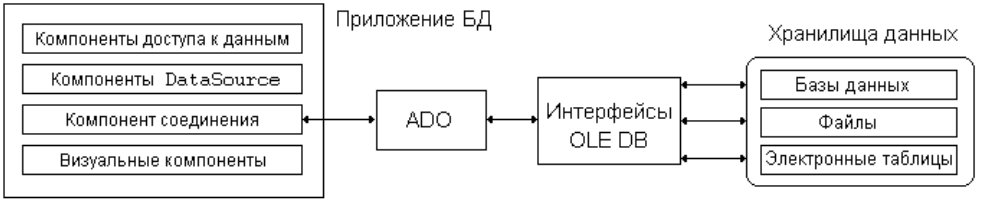


Рис. 22.1. Схема доступа к данным по технологии ADO

В соответствии с терминологией ADO, любой источник данных (базу данных, файл, электронную таблицу) называют *хранилищем данных*. Приложение взаимодействует с хранилищем данных с помощью *провайдера*. Для каждого типа хранилища данных используется свой провайдер ADO. Провайдер обеспечивает обращение к данным хранилища с запросами, интерпретацию возвращаемой служебной информации и результатов выполнения запросов для передачи их приложению.

Все объекты и интерфейсы ADO являются объектами и интерфейсами COM. Согласно спецификации OLE DB, в состав COM входит следующий набор объектов:

- *Command* (команда) — служит для обработки команд (обычно SQL-запросов);
- *Data Source* (источник *данных*) — используется для связи с провайдером данных;

- *Enumerator* (перечислитель) — служит для перечисления провайдеров ADO;
- *Error* (ошибка) — содержит информацию об исключениях;
- *Rowset* (набор рядов) — строки данных, являющиеся результатом выполнения команды;
- *Session* (сессия) — совокупность объектов, обращающихся к одному хранилищу данных;
- *Transaction* (транзакция) — управление транзакциями в OLE DB.

Мы привели состав объектов COM, чтобы получить общую картину о технологии ADO. В Delphi некоторые из этих объектов получили реализацию, и при необходимости мы будем рассматривать более подробно отдельные понятия.

В системе программирования Delphi компоненты, используемые для создания приложений по технологии ADO, расположены на странице **ADO** Палитры компонентов (рис. 22.2).

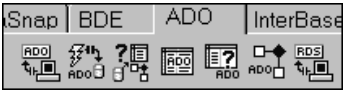


Рис. 22.2. Страница **ADO**

Охарактеризуем кратко назначение этих компонентов:

- `ADOConnection` — ADO-соединение, используется для установки соединения с ADO-источником данных и обеспечивает поддержку транзакций;
- `ADOCommand` — ADO-команды, используется для выполнения SQL-команд доступа к ADO-источнику данных без возвращения результирующего набора данных;
- `ADODataSet` — набор данных ADO, обеспечивает доступ к одной или более таблице ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом `ADOTable` через компонент `DataSource` аналогично тому, как используется компонент `DataSet`. Может использоваться в компонентах `ADOTable`, `ADOQuery`, `ADOStoredProc`;
- `ADOTable` — таблица ADO, обеспечивает доступ к одной таблице ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом `ADOTable` через компонент `DataSource`;
- `ADOQuery` — запрос ADO, позволяет выполнять SQL-команды для получения информации из ADO-источника данных и дает возможность другим компонентам управлять этими данными, связываясь с компонентом `ADOTable` через компонент `DataSource`;
- `ADOStoredProc` — хранимая процедура ADO, позволяет приложениям получать доступ к хранимым процедурам, используя интерфейс ADO;
- `RDSConnection` — RDS-соединение, служит для управления передачей объекта `Recordset` от одного процесса (компьютера) к другому. Компонент используется для создания серверных приложений.

Компонент ADOConnection может использоваться как посредник между данными и другими компонентами ADO, что позволяет более гибко управлять соединением.

Возможен вариант использования других компонентов ADO, соединяясь с источником данных напрямую, для чего компоненты ADO имеют свойство ConnectionString, с помощью которого могут создавать собственный канал доступа к данным.

Установление соединения

При использовании компонентов доступа к данным по технологии ADO (ADOCommand, ADODataSet, ADOTable, ADOQuery и ADOStoredProc) установление соединения с хранилищем данных можно выполнить двумя путями: с помощью свойства ConnectionString или с помощью компонента ADOConnection. Имя последнего задается в свойстве Connection компонентов доступа к данным. При этом во втором случае для компонента ADOConnection предварительно также с помощью его свойства ConnectionString нужно установить соединение с хранилищем данных.

Рассмотрим технологию установления соединения с хранилищем данных с помощью свойства ConnectionString. Это свойство представляет собой строку с параметрами соединения, отделяемыми друг от друга точкой с запятой. Предварительно соответствующий компонент доступа к данным (например, ADODataSet) или компонент соединения (ADOConnection) должен быть помещен на форму приложения. Настройка параметров соединения осуществляется в диалоге (рис. 22.3), открываемом двойным щелчком в строке ConnectionString свойства соответствующего компонента доступа к данным в окне Инспектора объектов.

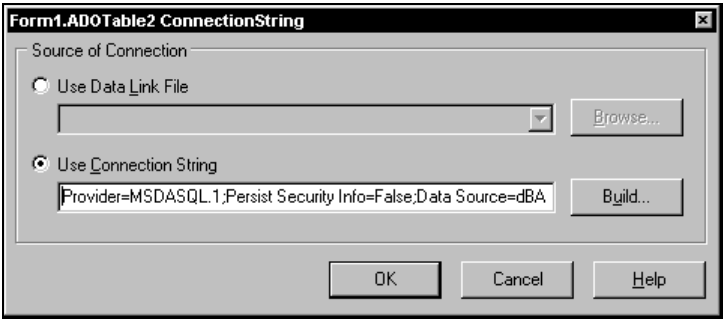


Рис. 22.3. Первое окно настройки строки соединения

При установке переключателя **Use Data Link File** можно выбрать из списка или найти (после нажатия кнопки **Browse**) *udl-файл связи с данными*. По умолчанию он расположен в папке c:\Program Files\Common Files\System\Ole DB\Data Links. По существу файлы связи с данными играют ту же роль, что и псевдонимы при использовании BDE. Они позволяют разработчику не связывать откомпилированные приложения с точным расположением хранилища данных. При перемещении хранилища данных в другое место достаточно исправить содержимое файла связи с данными.

При установке переключателя **Use Connection String** выполняются действия по *созданию строки соединения*. Рассмотрим их более подробно. Для продолжения выбранного

варианта диалога нужно нажать кнопку **Build**. В результате открывается диалоговое окно **Data Link Properties**, содержащее 4 вкладки. С помощью вкладки **Provider** (рис. 22.4) осуществляется выбор провайдера с учетом характера решаемой задачи. По умолчанию предлагается вариант **Microsoft OLE DB Provider for ODBC Drivers**.

Отметим, что после установки Microsoft ActiveX Data Objects в операционной системе доступны стандартные провайдеры ADO, обеспечивающие следующее:

- **Microsoft Jet 40 OLE DB Provider** — соединение с данными СУБД Microsoft Access;
- **Microsoft OLE DB Provider for Indexing Service** — доступ для чтения к ресурсам Microsoft Indexing Service;
- **Microsoft OLE DB Provider for Active Directory Service** — доступ к ресурсам службы каталогов Active Directory;
- **Microsoft OLE DB Provider for Internet Publishing** — доступ к ресурсам Microsoft FrontPage и Microsoft Internet Information Server;
- **Microsoft Data Shaping Service for OLE DB** — доступ к иерархическим наборам данных;
- **Microsoft OLE DB Simple Provider** — доступ к хранилищам данных, поддерживающим основные возможности OLE DB;
- **Microsoft OLE DB Provider for ODBC drivers** — доступ к данным для драйверов ODBC;

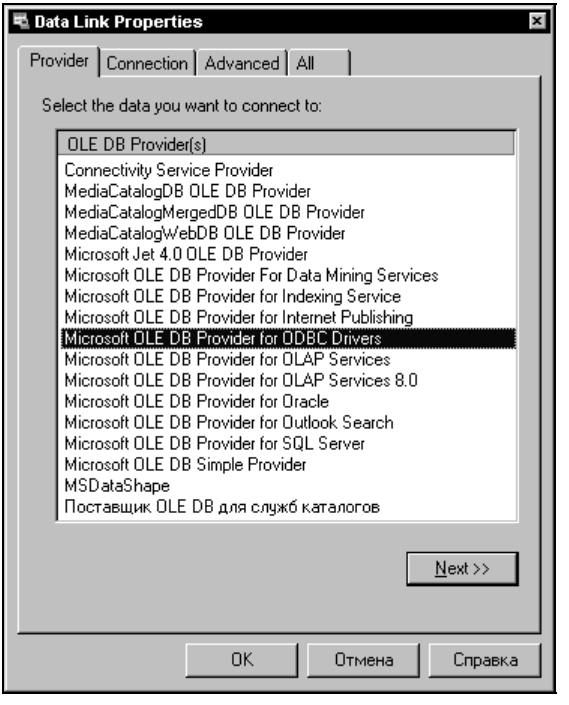


Рис. 22.4. Вкладка **Provider** окна настройки соединения

- **Microsoft OLE DB Provider for Oracle** — соединение с сервером Oracle;
- **Microsoft OLE DB Provider for SQL Server** — соединение с сервером Microsoft SQL Server.

При нажатии кнопки **Next >>** (см. рис. 22.4) происходит переход на вкладку **Connection**, содержимое которой несколько изменяется в зависимости от выбора провайдера. К примеру, вид вкладки **Connection** диалогового окна **Data Link Properties** для случая выбора провайдера **Microsoft Jet 4.0 OLE DB Provider** приведен на рис. 22.5.

На вкладке **Connection** можно указать имя базы данных, имя пользователя и пароль (для защищенных БД). Кроме того, нажав кнопку **Test Connection**, можно проверить правильность функционирования соединения. Далее можно нажатием кнопки **OK** установить строку соединения либо перейти на остальные две вкладки.

Рис. 22.5. Вкладка **Connection** окна настройки соединения

На вкладке **Advanced** в поле **Network Settings** задается уровень защиты при сетевом доступе к базе данных. В поле **Connect timeout** задается предельное время ожидания соединения в секундах. В списке **Access permissions** для определения прав доступа задается перечень допустимых операций: *Read* — только чтение; *ReadWrite* — чтение и запись; *Share Deny None* — нет запрета на чтение и запись; *Share Deny Read* — запрещено открытие для чтения; *Share Deny Write* — запрещено открытие для записи; *Share Exclusive* — эксклюзивное (монопольное) использование; *Write* — только запись.

На вкладке **All** диалогового окна настройки можно просмотреть и отредактировать параметры соединения, заданные с помощью других вкладок.

В случае использования компонента ADOConnection для активизации соединения после настройки достаточно установить свойству Connected этого компонента значение True или при выполнении приложения вызвать метод Open.

В случае использования любого из компонентов доступа к данным (ADODataSet, ADOTable, ADOQuery и ADOStoredProc) для активизации соединения после настройки используют свойство Active.

Управление соединением и транзакциями

Как следует из изложенного, установление соединения с хранилищем данных может осуществляться компонентами доступа к данным ADO через их свойство ConnectionString.

Кроме того, компоненты доступа к данным ADO могут соединяться с хранилищем данных через свойство Connection, связывающее их с компонентом соединения ADOConnection. При этом последний компонент связывается с хранилищем данных через свое свойство ConnectionString.

Достоинство применения компонента ADOConnection для связи компонентов доступа к данным с хранилищами данных состоит в том, что он позволяет управлять параметрами соединения и транзакциями.

С помощью свойства CoursorLocation типа TCursorLocation можно задать библиотеку, используемую курсором при соединении с хранилищем данных. Используемое по умолчанию значение clUseClient означает, что все данные располагаются и обрабатываются на компьютере-клиенте. При этом достигается наибольшая гибкость: возможны операции, которые могут не поддерживаться сервером. Тем не менее, операторы SQL все равно выполняются на сервере, а клиенту передаются результаты выполнения запроса.

Если свойству CoursorLocation задано значение clUseServer, то обработка данных ведется на сервере. Такое значение свойства целесообразно устанавливать в случае задания команд, возвращающих большой объем данных.

При использовании технологии ADO соединение может быть *синхронным* или *асинхронным*, что можно задать с помощью свойства ConnectOptions типа TConnectOption, который описан так:

```
type TConnectOption = (coConnectUnspecified, coAsyncConnect);
```

Значение coConnectUnspecified задает синхронное соединение, которое всегда ожидает результат последнего запроса, значение coAsyncConnect задает асинхронное соединение, при котором новые запросы выполняются, не ожидая ответа от предыдущих запросов.

Перед установлением соединения возникает событие OnWillConnect типа TWillConnectEvent, который описан так:

```
TWillConnectEvent = procedure(Connection: TADOConnection;
  var  ConnectionString, UserID, Password: WideString;
  var ConnectOptions: TConnectOption; var EventStatus: TEventStatus)
    of object;
```

Здесь: `Connection` указывает на соответствующий компонент соединения; `ConnectionString`, `Password`, `WideString` определяют строку соединения, имя и пароль пользователя соответственно; `ConnectOptions` указывает на синхронное или асинхронное соединение.

Параметр `EventStatus` указывает на успешность выполнения запроса на соединение. Тип этого параметра описан так:

```
type TEventStatus = (esOK,esErrorsOccured,esCantDeny,esCancel,esUnwantedEvent);
```

Здесь, например: `esOK` — соединение выполнено успешно; `esErrorsOccured` — ошибка при выполнении операции; `esCantDeny` — незаконченную операцию соединения нельзя отменить.

После установки соединения возникает событие `OnConnectComplete` типа `TConnectErrorEvent`, который описан так:

```
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
                               var EventStatus: TEventStatus) of object;
```

В случае возникновения ошибки в процессе соединения параметр `EventStatus` будет иметь значение `esErrorsOccured`, а параметр `Error` будет содержать объект ошибки ADO.

Замечание

Объекты ошибки ADO содержат информацию об ошибке, возникшей при выполнении операции над каким-либо объектом ADO. Разработчик может использовать методы объекта ошибки интерфейса `Error`, предоставляемого многими методами других объектов ADO. Укажем важные свойства объекта ошибки. Свойство `Description` возвращает описание ошибки, переданное из объекта. Свойство `SQLState` содержит текст команды, вызвавшей ошибку. Свойство `NativeError` возвращает код ошибки, переданный из объекта, в котором ошибка произошла.

При разрыве соединения возникает событие `OnDisconnect` типа `TDisconnectEvent`, который описан так:

```
TDisconnectEvent = procedure(Connection: TADOConnection;
   var EventStatus:  TEventStatus) of object;
```

Управление транзакциями осуществляется с помощью методов и свойств компонента `ADOConnection`.

Для *запуска* транзакции используется функция `BeginTrans`, возвращающая целое значение — уровень вложенности новой транзакции. В случае успешного запуска транзакции свойство `InTransaction` принимает значение `True`, указывающее на то, что компонент соединения находится в транзакции.

Для *завершения* транзакции и сохранения внесенных изменений в хранилище данных используется метод `CommitTrans`. При успешном его выполнении свойство `InTransaction` принимает значение `False`.

Для *отката* транзакции служит метод `RollbackTrans`. При его успешном выполнении отменяются все изменения, внесенные в ходе транзакции, а свойство `InTransaction` принимает значение `True`.

Для управления запуском транзакций, оставшихся незавершенными, служит свойство `Attributes` типа `TXactAttributes`. Оно принимает два значения: `xaCommitRetaining` — незавершенная транзакция начинается при подтверждении предыдущей транзакции; `xaAbortRetaining` — незавершенная транзакция начинается при отмене предыдущей транзакции.

Компоненты доступа к данным

Перечень и назначение компонентов работы по технологии ADO мы приводили в начале главы. Большинство из них имеют аналоги компонентов в технологиях BDE и dbExpress (табл. 22.1). В этой таблице компоненты, используемые в технологии dbExpress, аналогами можно назвать достаточно условно, т. к. большинство из них поддерживают однонаправленные наборы данных.

Таблица 22.1. *Соответствие компонентов в технологиях ADO, dbExpress и BDE*

Компонент ADO	Компонент dbExpress	Компонент BDE
`ADOTable`	`SQLTable`	`Table`
`ADOQuery`	`SQLQuery`	`Query`
`ADOStoredProc`	`SQLStoredProc`	`StoredProc`
`ADOConnection`	`SQLConnection`	`Database`
`ADODataSet`	`SQLDataSet`, `SQLSimpleDataSet`	`Table`, `Query`, `StoredProc`
`ADOCommand`	—	—
`RDSConnection`	—	—

Стандартные компоненты доступа к данным в ADO наследуют механизм доступа от родительского класса `TCustomADODataSet`. Поэтому важнейшие свойства и методы этого класса во многом определяют поведение компонентов доступа к данным в ADO. Коротко охарактеризуем их.

К числу основных свойств названного класса можно отнести свойства, устанавливающие параметры обмена с хранилищем данных. Значения их нужно задать до открытия набора данных.

Тип блокировки записей в наборе данных определяет свойство `LockType` типа `TADOLockType`, который описан так:

```
type TADOLockType = (ltUnspecified, ltReadOnly, ltPessimistic,
                     ltOptimistic, ltBatchOptimistic);
```

Здесь, например:

- `ltUnspecified` — блокировка задается источником данных;
- `ltReadOnly` — набор данных открывается в режиме "только для чтения";
- `ltPessimistic` — блокировка на время редактирования до момента подтверждения.

Как отмечалось, местоположение курсора задает свойство `CursorLocation` *(см. предыдущий подраздел)*. При работе с клиентским курсором важную роль играет следующее свойство.

Передаваемые серверу данные определяет свойство `MarshalOptions` типа `TMarshalOption`, который описан так:

```
type TMarshalOption = (moMarshalAll, moMarshalModifiedOnly);
```

Здесь:

- `moMarshalAll` — возврат всех записей локального набора данных серверу;
- `moMarshalModifiedOnly` — возврат только измененных записей.

Тип курсора определяет свойство `CursorType` типа `TCursorType`, который описан так:

```
type TCursorType = (ctUnspecified, ctOpenForwardOnly,
                    ctKeyset, ctDynamic, ctStatic);
```

Здесь, например:

- `ctUnspecified` — тип курсора не задан и определяется возможностями источника данных;
- `ctOpenForwardOnly` — однонаправленный курсор, используемый для одиночного прохода по всем записям;
- `ctKeyset` — двунаправленный курсор, не отображающий добавленные или удаленные другими пользователями записи;
- `ctStatic` — двунаправленный курсор, не учитывающий изменения записей другими пользователями.

Набор данных может быть открыт с помощью свойства `Active` или метода `Open`. На стороне клиента записи из набора данных хранятся в буфере, размер которого может быть получен с помощью свойства `CacheSize` типа `Integer`.

При необходимости с помощью свойства `BlockReadSize` типа `Integer` можно организовать передачу записей в виде блоков. Кроме того, с помощью свойства `MaxRecords` типа `Integer` можно ограничить размер набора данных. По умолчанию блочная пересылка не используется, а число записей в наборе данных не ограничено.

Текущее *состояние записи* набора данных определяет свойство `RecordStatus` типа `TRecordStatusSet`, который описан так:

```
type TRecordStatus = (rsOK, rsNew, rsModified, rsDeleted, rsUnmodified,
                      rsInvalid, rsMultipleChanges, rsPendingChanges,
                      rsCanceled, rsCantRelease, rsConcurrencyViolation,
                      rsIntegrityViolation, rsMaxChangesExceeded,
                      rsObjectOpen, rsOutOfMemory, rsPermissionDenied,
                      rsSchemaViolation, rsDBDeleted);
```

Здесь, например:

- `rsOK` — запись успешно изменена;
- `rsNew` — новая запись вставлена;
- `rsModified` — запись изменена;

- rsDeleted — запись удалена;
- rsUnmodified — запись осталась без изменений.

Дадим сравнительно развернутую характеристику компонентам, используемым в технологии ADO для доступа к данным.

Доступ к таблицам

Для обеспечения доступа к таблицам хранилищ данных по технологии ADO служит компонент ADOTable. Для установления соединения с хранилищем данных этого компонента через провайдеры ADO служит свойство ConnectionString или Connection, как описано ранее. Для управления набором данных таблицы в приложение включают компонент источника данных DataSource. При этом свойству DataSet этого компонента в качестве значения задается имя компонента ADOTable. Для отображения данных таблицы к источнику данных подключаются различные компоненты отображения, к примеру, DBGrid.

После установления связи компонента ADOTable с хранилищем данных с помощью свойства TableName типа WideString задается имя таблицы. Не все провайдеры ADO допускают непосредственный доступ к таблицам, поэтому может потребоваться доступ с помощью SQL-запроса. Вариант доступа к данным таблицы определяет свойство TableDirect типа Boolean. Если оно имеет значение False (по умолчанию), то компонент ADOTable автоматически генерирует SQL-запрос для доступа к таблице, в противном случае выполняется непосредственный доступ к данным таблицы.

Рассматриваемый нами компонент по своим возможностям и технике работы с ним во многом схож с компонентом Table. Здесь также с помощью редактора полей можно задавать свойства отдельных полей. При этом имеется ограничение, состоящее в том, что в компонентах ADO нельзя работать со словарями. Поэтому свойства полей требуется задавать вручную. Кроме того, у драйверов ADO имеются ограничения по работе с отдельными типами полей, в частности, с графическими.

Для программирования действий по работе с хранилищем данных с помощью рассматриваемого компонента используются аналогичные средства, как и в случае компонента Table. В частности, для навигации по табличному набору данных используются методы First, Last, Next и Prior. Для поиска записей используются методы Find, Seek и Locate.

Выполнение запросов

Для выполнения SQL-запросов при использовании технологии ADO служит компонент ADOQuery. По функциональным возможностям и технике применения этот компонент во многом подобен компоненту Query. Установка соединения с хранилищем данных, свойства и методы фильтрации и поиска аналогичны используемым для компонента ADOTable.

Текст запроса задается с помощью свойства SQL типа TStrings.

С помощью свойства Parameters типа TParameters определяются параметры запроса.

Открытие набора данных может быть выполнено с помощью свойства Active типа Boolean или с помощью метода Open. Если же запрос не должен возвращать данных, то для открытия набора данных можно вызвать метод ExecSQL.

Вызов хранимых процедур

Для вызова хранимых процедур по технологии ADO служит компонент `ADOStoredProc`. По своим возможностям и технике использования он подобен своему аналогу из BDE. Установка соединения с хранилищем данных и управление набором данных аналогичны используемым для компонента `ADOTable`. Для отображения данных (выходных параметров) к источнику данных также подключаются компоненты отображения.

Для задания имени хранимой процедуры служит свойство `ProcedureName` типа `WideString`.

Свойство `Parameters` типа `TParameters` служит для определения входных параметров процедуры. А именно после задания имени хранимой процедуры в свойстве `Parameters` отображаются входные параметры процедуры. Их можно просмотреть с помощью Инспектора объектов.

Выходные параметры хранимой процедуры являются объектами полей рассматриваемого компонента `ADOStoredProc`. Чтобы просмотреть и изменить их значения, достаточно с помощью контекстного меню компонента `ADOStoredProc` вызвать редактор полей **Fields Editor**. С помощью команд редактора можно легко изменить состав полей (выходные параметры), отображаемых в наборе данных компонента `ADOStoredProc`.

Компонент *ADODataSet*

Компонент `ADODataSet` служит для представления набора данных из хранилища данных ADO. По своим функциональным возможностям компонент в определенной мере аналогичен компонентам `SQLDataSet` и `SQLSimpleDataSet`, используемым в технологии dbExpress. Этот компонент позволяет получать данные из таблиц, SQL-запросов, хранимых процедур, файлов и т. д.

Тип команды, указываемой в свойстве `CommandText`, определяет свойство `CommandType` типа `TCommandType`, который определен так:

```
TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc,
                cmdFile, cmdTableDirect)
```

Здесь:

- `cmdUnknown` — тип не известен;
- `cmdText` — текст SQL-оператора или вызова процедуры (значение по умолчанию);
- `cmdTable` — имя таблицы;
- `cmdStoredProc` — имя хранимой процедуры;
- `cmdFile` — имя сохраненного файла набора данных;
- `cmdTableDirect` — имя таблицы, все поля которой возвращаются.

Значение свойства `CommandText` типа `WideString` определяет команду, которая выполняется. В качестве команды может быть задана строка с SQL-оператором, имя таблицы или имя хранимой процедуры. При необходимости могут быть заданы параметры с помощью свойства `Parametrs`. Выполнение команды, заданной с помощью свойства `CommandText`, происходит при открытии набора данных.

Установка соединения с хранилищем данных и управление набором данных аналогичны используемым для компонента ADOTable. Для отображения данных к источнику данных (компоненту DataSource) также подключаются компоненты отображения.

Команды ADO

Для выполнения команд предназначен компонент ADOCommand, являющийся реализацией объекта ADO Command в среде Delphi. Этот компонент служит для выполнения команд, не возвращающих результаты, например, SQL-операторов языка определения данных или хранимых процедур. Для выполнения хранимых процедур и SQL-запросов, возвращающих результаты в виде наборов данных, целесообразно применять компоненты доступа к данным ADODataSet, ADOQuery и ADOStoredProc.

Рассмотрим важнейшие свойства и методы компонента ADOCommand. Отметим, что большинство свойств компонента ADOCommand в Delphi эквивалентно свойствам объекта ADO Command.

Свойство CommandObject типа _Command предоставляет непосредственный доступ к базовому объекту команды ADO Command.

Тип команды, указываемой в свойстве CommandText, определяет свойство CommandType (см. описание компонента ADODataSet). Для свойства CommandType объекта ADOCommand значения cmdFile и cmdTableDirect являются недопустимыми.

Напомним, что значение свойства CommandText типа WideString определяет команду, которая выполняется с помощью метода Execute. В качестве команды может быть задана строка с SQL-оператором, имя таблицы или имя хранимой процедуры. При необходимости могут быть заданы параметры с помощью свойства Parametrs.

Текущее состояние компонента ADOCommand (объекта Command) определяет свойство States типа ObjectStates, который описан так:

```
TObjectState = (stClosed, stOpen, stConnecting, stExecuting, stFetching);
```

Здесь:

- stClosed — объект Command не активен и не соединен с хранилищем данных;
- stOpen — объект Command не активен, соединен с хранилищем данных;
- stConnecting — объект Command соединяется с хранилищем данных;
- stExecuting — объект Command выполняет команду;
- stFetching — объект Command получает данные от хранилища данных.

Условия выполнения команды ADO определяет свойство ExecuteOptions типа TExecuteOptions, который описан так:

```
TExecuteOption = (eoAsyncExecute, eoAsyncFetch, eoAsyncFetchNonBlocking,
                  eoExecuteNoRecords);
```

Здесь:

- eoAsyncExecute — асинхронное выполнение команды;
- eoAsyncFetch — асинхронная передача данных;

- `eoAsyncFetchNonBlocking` — асинхронная передача данных без блокирования потока;
- `eoExecuteNoRecords` — выполнение команды ADO без возвращения данных.

Выполнение команды ADO осуществляется с помощью одной из трех перегружаемых версий метода `Execute` с различными наборами параметров:

- `function Execute: _Recordset; overload;` — такое объявление метода используется при выполнении команды без параметров;
- `function Execute(const Parameters: OleVariant): _Recordset; overload;` — в этом объявлении параметр `Parameters` определяет параметры команды;
- `function Execute(var RecordsAffected: Integer; const Parameters: OleVariant): _RecordSet; overload;` — здесь параметр `RecordsAffected` возвращает общее число обработанных запросом записей.

Пример приложения

Рассмотрим пример (рис. 22.6) приложения для работы с БД, использующего технологию ADO. Пусть назначением приложения является организация доступа к различным таблицам из базы данных, их редактирование и сохранение внесенных изменений на сервере.

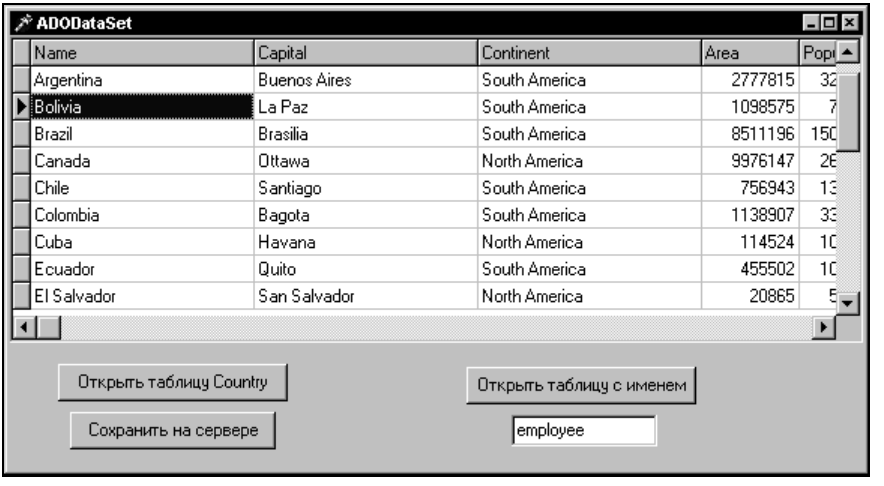


Рис. 22.6. Вид формы при выполнении приложения

Основными компонентами приложения, используемыми для установления соединения с базой данных и отображения данных, являются следующие: `ADOConnection1`, `ADODataSet1`, `DataSource1` и `DBGrid1`. Кроме того, имеется ряд вспомогательных компонентов, служащих для управления работой приложения.

Соединение с хранилищем данных компонента `ADODataSet1` выполнено через его свойство `Connection`, указывающее на имя `ADOConnection1` компонента соединения. Для компонента `ADOConnection1` с помощью его свойства `ConnectionString` установлено соединение с хранилищем данных. При этом использован файл связи с данными. Для

наглядности в процедуре обработки события создания формы приведен оператор задания нужного значения свойству `ConnectionString` компонента `ADOConnection1`.

Для компонента `DataSource1` его свойству `DataSet` установлено значение `ADODataSet1`. Для компонента `DBGrid1`, используемого для отображения набора данных, его свойству `DataSource` установлено значение `DataSource1`.

В листинге 22.1 приведен код модуля формы для решения поставленной задачи.

Листинг 22.1. Пример приложения для работы по технологии ADO

```
unit ADODataSet;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, DB, ADODB, Grids, DBGrids, ExtCtrls;

type
  TForm1 = class(TForm)
    ADODataSet1:    TADODataSet;
    DataSource1:    TDataSource;
    DBGrid1:        TDBGrid;
    ADOConnection1: TADOConnection;
    Panel1:         TPanel;
    UpdateButton:   TButton;
    GetTableButton: TButton;
    Button1:        TButton;
    Edit1:          TEdit;
    procedure Form1Create(Sender: TObject);
    procedure Form1CloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure UpdateButtonClick(Sender: TObject);
    procedure GetTableButtonClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    private
public
    procedure UpdateData;
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
const
  BaseFileName = 'EMPLOYEE.ADTG';

procedure TForm1.UpdateData;
begin
```

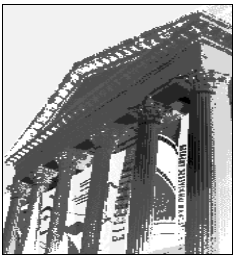
```
  { Обновление данных на сервере }
  ADODataSet1.UpdateBatch;
end;

procedure TForm1.Form1Create(Sender: TObject);
begin
  ADOConnection1.ConnectionString := 'FILE NAME=' + DataLinkDir +
     '\DBDEMOS.UDL';
  ADODataSet1.Open;
end;

procedure TForm1.UpdateButtonClick(Sender: TObject);
begin
  UpdateData;
end;

procedure TForm1.Form1CloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if ADODataSet1.Active then
  try
    { Обновление данных на сервере при закрытии формы }
    UpdateData
  except
    on E: Exception do
    begin
      Application.HandleException(Self);
      CanClose := MessageDlg('Данные не сохранены/обновлены, выйти?',
        mtConfirmation, mbYesNoCancel, 0) = mrYes;
    end;
  end;
end;

procedure TForm1.GetTableButtonClick(Sender: TObject);
begin
  { Повторное открытие табличного набора данных 'Country'}
  ADODataSet1.Close;
  ADODataSet1.CommandType := cmdTable;
  ADODataSet1.CommandText := 'Country';
  ADODataSet1.Open;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ADODataSet1.Close;
  ADODataSet1.CommandType := cmdTable;
  ADODataSet1.CommandText := Edit1.Text;
  try
    ADODataSet1.Open;
  except
    on E: Exception do
```

```
    begin
      MessageDlg('Исправьте имя таблицы', mtError,[mbOK],0);
      ADODataSet1.Close;
    end;
  end;
end;
end.
```

Первоначально при запуске приложения автоматически открывается таблица с именем `Country`. При необходимости открыть другую таблицу для просмотра и редактирования ее имя следует ввести в однострочном редакторе (компонент `Edit1`) и затем нажать кнопку **Открыть таблицу с именем**.

Обновление внесенных в открытую таблицу изменений происходит при нажатии кнопки **Сохранить на сервере** или автоматически при закрытии формы приложения.

ГЛАВА 23



Создание и просмотр отчетов с помощью Rave Reports

Отчет представляет собой печатный документ, содержащий данные, аналогичные получаемым в результате выполнения запроса к БД или из некоторого другого источника — электронной таблицы, сообщения электронной почты, текстового документа и др. Можно выделить следующие виды отчетов:

- простой отчет;
- отчет с группированием данных;
- отчет для таблиц, связанных отношением "главный — подчиненный";
- составной отчет, объединяющий несколько разных отчетов.

Характеристика генератора отчетов

В Delphi 7 для создания отчетов предназначен генератор отчетов Rave Reports 5.0 фирмы Nevrona, компоненты которого, предназначенные для управления отчетами, размещены на странице **Rave** Палитры компонентов. Назначение компонентов указано в *главе 14*. Генератор отчетов имеет развитые средства визуального проектирования отчетов.

Из состава Палитры компонентов изъята страница **QReport**, содержащая набор компонентов, предназначенных для создания отчетов. Как отмечалось, для обеспечения обратной совместимости компоненты страницы **QReport** размещены в пакете dclqrt70.bpl в папке \Delphi 7\Bin и могут быть установлены в Палитру компонентов.

Генератор отчетов Rave Reports позволяет создавать отчеты для обычных приложений и для приложений баз данных. При этом обеспечивается возможность создания отчетов для приложений баз данных, использующих различные механизмы доступа к данным BDE, ADO и dbExpress.

Компоненты, расположенные на странице **Rave** Палитры компонентов, предназначены для управления отчетами в приложении. Важнейшими из них являются компоненты `RvProject` и `RvSystem`. Так, компонент `RvProject` типа `TRvProject` задает представление проекта отчета в приложении Delphi, а компонент `RvSystem` служит для управления от-

четом и, к примеру, позволяет задать печать, предварительный просмотр или передачу отчета в файл.

Отчет размещается в файле с расширением rav, который представляет проект создаваемого отчета и содержит информацию об отчете, параметры оформления страниц отчета и др. В состав проекта отчета может входить один или несколько отчетов, в каждом из которых может быть произвольное число страниц и глобальных страниц. На страницах отчета и на глобальных страницах могут размещаться графические и текстовые объекты, связанные с источниками данных приложения Delphi.

Визуальный конструктор отчетов Rave Reports 5.0 повышает удобство и упрощает непосредственную разработку отчетов. Вызвать визуальный конструктор отчетов можно тремя путями: с помощью команды **Tools | Rave Designer** меню Delphi, двойным щелчком мыши на компоненте `RvProject` или с помощью команды **Rave Visual Designer** контекстного меню компонента `RvProject`.

Кроме того, в состав генератора отчетов входит ядро, которое обеспечивает управление отчетом, предварительный просмотр и отправку на печать. Код ядра при компиляции помещается в приложение, тем самым обеспечивается автономность последнего.

Визуальное конструирование отчетов

Визуальный конструктор отчетов Rave Reports 5.0 служит для повышения удобства и упрощения непосредственной разработки отчетов. С его помощью можно быстро создать новый отчет, глобальную страницу, страницу отчета, настроить параметры отчета, выполнить подключение источника данных и др.

Интерфейс визуального конструктора

Вызов визуального конструктора отчетов можно выполнить с помощью команды **Tools | Rave Designer** меню Delphi, двойным щелчком мыши на компоненте `RvProject` или с помощью команды **Rave Visual Designer** контекстного меню компонента `RvProject`. Возможный вид окна Rave Reports 5.0 приведен на рис. 23.1.

Визуальный конструктор отчетов имеет практически стандартный интерфейс приложений Windows. В верхней части окна визуального конструктора расположено меню. Под ним находится панель инструментов, кнопки которой дублируют основные команды визуального конструктора. В правой верхней части конструктора размещена многостраничная панель инструментов, на вкладках которой содержатся компоненты, используемые при создании отчетов, а также элементы, управляющие параметрами объектов отчета (цвет линий, шрифт, выравнивание) и масштабом отображения отчета в окне.

Инспектор компонентов отчета, расположенный в левой части окна визуального конструктора, позволяет выполнить настройку свойств компонентов отчета, которые выбираются в дереве проекта отчета. В нижней части Инспектора компонентов отчета может отображаться подсказка для выбранного свойства. Управление отображением этой подсказки и подсветкой измененных свойств компонентов выполняется с помощью контекстного меню Инспектора компонентов отчета.

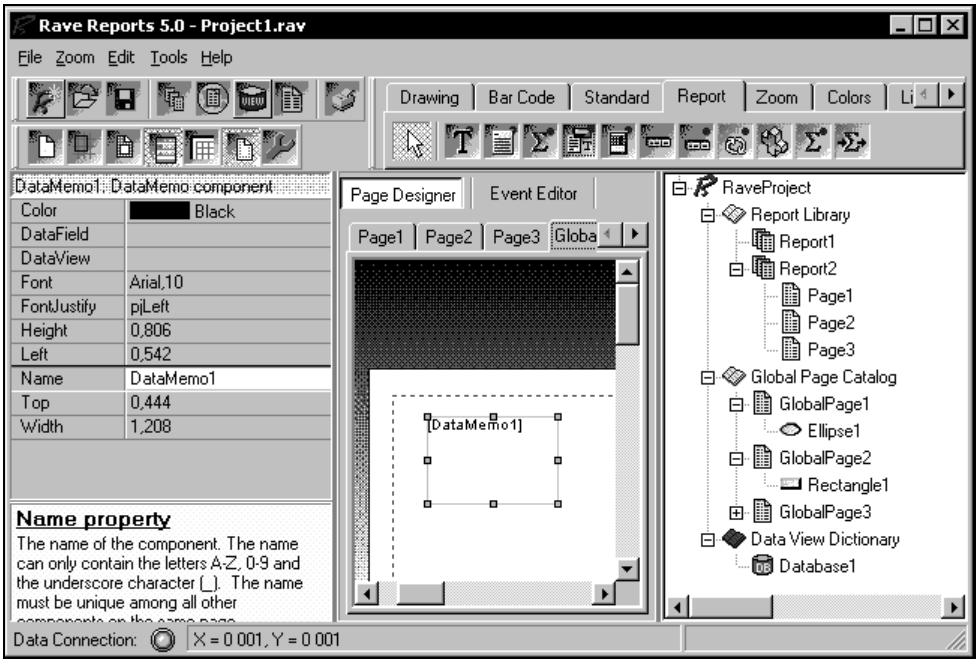


Рис. 23.1. Вид окна Rave Reports 5.0

В центральной части окна визуального конструктора расположен блокнот с двумя вкладками: **Page Designer** (Конструктор страниц) и **Event Editor** (Редактор событий). С помощью вкладки **Page Designer** можно добавлять, удалять и настраивать компоненты на отдельных страницах отчета. На этой вкладке имеется еще один блокнот, каждая из вкладок которого содержит одну страницу отчета. С помощью вкладки **Event Editor** создаются обработчики событий для отчетов, страниц, элементов оформления и компонентов отчета.

Дерево проекта отчета, размещенное в правой части окна визуального конструктора, отображает состав компонентов открытого проекта отчета. С помощью двойного щелчка удобно выбирать компоненты проекта отчета для отображения в центральной части окна и настройки свойств в Инспекторе компонентов.

Состав проекта отчетов

В дереве проекта отчетов RaveProject (см. рис. 23.1) имеются следующие три составляющие:

- **Report Library** (Библиотека отчетов проекта) — содержит все отчеты проекта;
- **Global Page Catalog** (Каталог глобальных страниц) — содержит перечень страниц, являющихся общими для всего проекта;
- **Data View Dictionary** (Словарь просмотров данных) — содержит объекты соединения с данными из внешних источников.

Рассмотрим более подробно каждую из названных составляющих проекта отчетов.

Библиотека отчетов проекта содержит все отчеты, входящие в состав проекта. Отдельный отчет в составе проекта включает произвольное число страниц, содержащих компоненты с данными, текстом и графикой, а также компоненты, повышающие наглядность оформления отчета.

При работе с проектом отчетов выделяют текущий отчет. При открытии проекта отчетов текущим становится первый отчет в составе проекта отчетов. При необходимости двойным щелчком мыши в дереве проекта отчетов можно установить текущим произвольный отчет. При выходе из среды визуального конструктора текущий отчет в составе проекта сохраняется и может использоваться при работе с отчетом с помощью компонента `RvProject` в приложении Delphi.

К проекту отчета можно добавлять новые отчеты с помощью команды **File | New Report** визуального конструктора. К текущему отчету можно добавлять страницы с помощью команды **File | New Report Page**. Для удаления отчета, страницы отчета или глобальной страницы достаточно выделить отчет или страницу отчета в дереве проекта отчетов и нажать клавишу <Delete>.

В отчет может входить произвольное число страниц. Перед предварительным просмотром и печатью отчета можно выбрать произвольное подмножество страниц отчета и добавить нужные страницы из каталога глобальных страниц. В отчете для управления составом и очередностью печати страниц предназначено свойство `PageList`. Если значение этого свойства не определено, то печатаются все страницы отчета в порядке их следования в дереве проектов отчета. Для задания значения свойства `PageList` используется редактор списка страниц Page List Editor (рис. 23.2), позволяющий сформировать список страниц для печати.

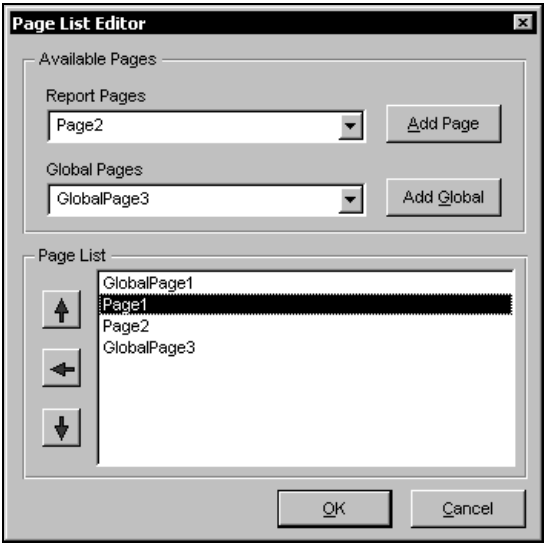


Рис. 23.2. Окно редактора Page List Editor

В проекте отчетов RaveDemo.rav, расположенном в папке \Borland\Delphi7 \Rave\Demos, имеется библиотека, содержащая множество различных отчетов. Напри-

мер, в ее составе имеются такие отчеты, как MultiPageReport (многостраничный отчет), BarCodes (штрихкоды), MasterDetailReport (отчет "мастер — детальный"). При необходимости полезно использовать отчеты из этой библиотеки в качестве шаблона при создании аналогичных отчетов.

Каталог глобальных страниц содержит страницы, которые являются доступными в любом отчете из состава библиотеки отчетов открытого проекта. Это позволяет получить стандартное оформление для ряда страниц отчета, таких как титульный лист, рамка для дипломных проектов и отчетов о научно-исследовательской работе и т. п.

Напомним, что имеющуюся в каталоге глобальную страницу можно добавить в список страниц отчета для печати с помощью редактора списка страниц (см. рис. 23.2).

Для открытого проекта отчетов в состав каталога пустую глобальную страницу можно добавить, выполнив команду **File | New Global Page**. После этого глобальную страницу можно редактировать и добавлять в состав страниц любого из отчетов.

Словарь просмотров данных содержит объекты доступа к данным из внешних источников. Создание новых объектов выполняется после задания команды **File | New Data Object**. При этом открывается диалоговое окно **Data Connections** (рис. 23.3), в котором для выбора предлагаются следующие типы объектов:

- **Data Lookup Security Controller** (Контроллер безопасности поиска данных) — обеспечивает аутентификацию пользователей по имени и паролю;
- **Database Connection** (Соединение с базой данных) — устанавливает соединение с внешним источником данных для требуемой технологии доступа (ADO, BDE, dbExpress);
- **Direct Data View** (Прямой просмотр данных) — создает просмотр данных для активного соединения с источником данных;
- **Driver Data View** (Просмотр данных с помощью драйвера) — создает просмотр данных на основе уже имеющегося в словаре соединения;
- **Simple Security Controller** (Простой контроллер безопасности) — задает список пользователей для возможной организации доступа в отчетах.

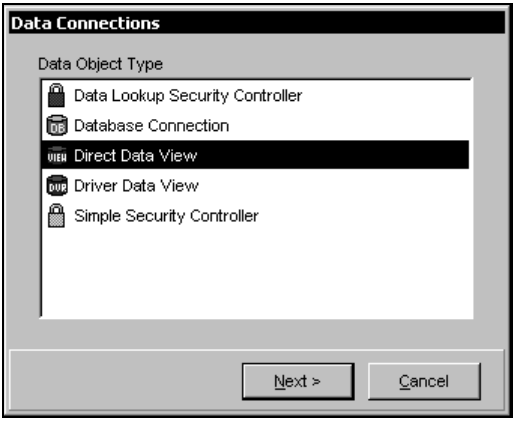


Рис. 23.3. Диалоговое окно **Data Connections**

Объекты в составе словаря просмотров данных доступны для всех отчетов, входящих в состав проекта отчетов. Названные типы объектов используются преимущественно при создании отчетов для приложений баз данных.

Редактор событий

В проекте любому отчету, странице, элементу оформления или компоненту отчета можно назначить один или несколько обработчиков событий. Сделать это можно с помощью редактора событий, расположенного на вкладке **Event Editor** (рис. 23.4) блокнота в центральной части окна визуального конструктора.

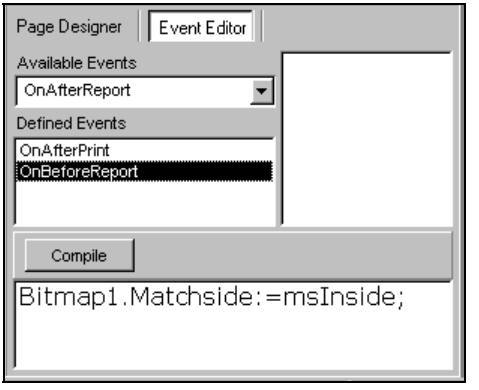


Рис. 23.4. Вкладка **Event Editor**

Создание обработчика событий выполняется следующим образом. В дереве проекта отчетов выбирается текущим отчет, страница или компонент, для которого требуется создать обработчик. Далее в списке **Available Events** выбирается одно из доступных для обработки событий. Это событие оказывается в списке **Defined Events** (Определенные события) и автоматически создается обработчик этого события. Теперь для этого события следует задать собственно код обработки события в нижней части окна редактора событий и нажать кнопку **Compile**. При этом происходит компиляция кода обработчика с выдачей сообщения о завершении компиляции или с сообщением об ошибке. При необходимости можно выбрать другое событие в списке **Available Events** и назначить ему свой обработчик.

При написании кода обработки событий используется синтаксис, близкий к синтаксису языка Object Pascal, допускается применение некоторых методов Delphi. К примеру, на рис. 23.4 приведен код обработчика события `OnBeforeReport`, с помощью которого свойству `Matchside` компонента `Bitmap1` присваивается значение `msInside`.

Компоненты, представленные на многостраничной панели инструментов

Укажем назначение вкладок многостраничной панели инструментов, расположенной в правой верхней части окна визуального конструктора отчетов.

Компоненты, используемые при создании отчетов, расположены на следующих вкладках:

- **Drawing** — графические компоненты, повышающие наглядность оформления отчета;
- **Bar Code** — компоненты, задающие различные варианты штрихкода;
- **Standard** — компоненты, задающие отображение текста и графики;
- **Report** — компоненты, предназначенные для отображения данных из внешних источников.

Элементы, управляющие параметрами объектов отчета (цвет линий, шрифт, выравнивание) и масштабом отображения отчета, расположены на следующих вкладках:

- **Zoom** — компоненты, управляющие масштабом отображения текущей страницы;
- **Colors** — компоненты, задающие цвета для графических компонентов и страниц;
- **Lines** — компоненты, задающие стиль и толщину линий графических компонентов;
- **Fills** — компоненты, задающие заливку графических компонентов;
- **Fonts** — компоненты, задающие параметры шрифта для текста;
- **Alignment** — компоненты, задающие выравнивание графических компонентов на странице.

Добавление компонентов в отчет выполняется традиционным для систем визуального программирования способом: щелчком мыши требуемый компонент выбирается на палитре и повторным щелчком мыши размещается в нужном месте страницы отчета. Дадим краткую характеристику компонентам, расположенным на различных страницах панели инструментов.

Страница **Drawing** многостраничной панели инструментов содержит компоненты, обеспечивающие создание графических объектов.

Компоненты `Line`, `Hline` и `Vline` служат для рисования универсальной, горизонтальной и вертикальной линий соответственно. Все они имеют набор свойств, определяющих длину и местоположение на странице отрезка линии. Для компонента `Line` отрезок можно развернуть произвольным образом визуально (с помощью мыши). Для двух других компонентов поворот отрезка также возможен, но не визуально, а только путем изменения свойств компонентов.

Компоненты `Rectangle` и `Square` служат для рисования прямоугольника и квадрата, а компоненты `Ellipse` и `Circle` — для рисования эллипса и круга соответственно. Эти четыре компонента также имеют набор свойств, определяющих их размеры и местоположение на странице.

Страница **Bar Code** многостраничной панели инструментов содержит компоненты, задающие штрихкоды, которые реализуют различные стандарты.

Компонент `PostNetBarCode` соответствует коду PostNet почтовой службы США, задающему код адреса.

Компонент `I2of5BarCode` соответствует перемежающемуся коду I2of5 и служит для задания числовых последовательностей, в которых цифры поочередно кодируются штрихами и пробелами.

Компонент `Code39BarCode` соответствует коду Code39 и служит для кодирования цифр, заглавных латинских букв и ряда символов с помощью штрихов и пробелов.

Компонент `Code128BarCode` соответствует коду Code128 и служит для представления первых 128 символов кода ASCII.

Компонент `UPCBarCode` соответствует коду UPC (Universal Product Code) и служит для маркировки продуктов, может включать до 12 цифр.

Компонент `EANBarCode` соответствует коду EAN (European Article Numbering system), аналогичен коду UPC, может включать до 13 цифр.

Для всех компонентов кодируемое значение задается свойством `Text`. Оно может быть также загружено из базы данных с помощью свойств `FileLink` и `DataLink`.

С помощью свойства `BarCodeRotation` штрихкод можно поворачивать с дискретностью 90°.

Страница **Standard** многостраничной панели инструментов содержит компоненты, задающие отображение текста и графики.

Компонент `text` служит для отображения однострочного текста. Сам отображаемый текст задается путем редактирования в Инспекторе компонентов отчета значения свойства `Text` этого компонента. Непосредственно на странице отчета отображаемый текст редактировать нельзя.

Компонент `memo` служит для отображения многострочного текста. Отображаемый компонентом текст определяется свойством `Text`, которое в Инспекторе компонентов отчета визуального конструктора связано с многострочным текстовым редактором Memo Editor.

Остальные свойства компонентов `text` и `memo` позволяют настроить цвет, шрифт, выравнивание местоположения текста и др.

Компонент `section` (секция) является невизуальным и предназначен для объединения в одну группу нескольких компонентов, к примеру, строк текста и изображений внутри заголовка отчета. Все размещенные внутри секции компоненты можно по отдельности перемещать в пределах секции, а при перемещении секции по странице вместе с секцией перемещаются и они. Выравнивание компонентов внутри секции осуществляется относительно ее границ.

Компонент `Bitmap` служит для отображения растровых изображений, хранящихся в файлах формата BMP. Отображаемый компонентом рисунок определяется с помощью свойства `Image`, которое в Инспекторе компонентов отчета визуального конструктора связано с диалогом загрузки файла изображения.

Компонент `MetaFile` служит для демонстрации изображений, хранящихся в файлах форматов EMF и WMF. Отображаемое компонентом изображение определяется с помощью свойства `Image`, которое в Инспекторе компонентов отчета визуального конструктора также связано с диалогом загрузки файла изображения. При этом файлы формата WMF таким способом загрузить нельзя, для этой цели нужно использовать свойство `FileLink` или `DataField`.

Изображения, выводимые с помощью компонентов `Bitmap` и `MetaFile`, можно масштабировать. При этом учитывается свойство `MatchSide`. При значениях этого свойства

msHeight или msWidth (по умолчанию) высота или ширина изображения соответственно устанавливаются равными высоте (свойство Height) или ширине (свойство Width) исходной области, выделенной под изображение, а ширина или высота изображения соответственно устанавливаются с сохранением пропорций изображения. Если свойство MatchSide имеет значение msBoth, то ширина и высота изображения устанавливаются равными ширине и высоте выделенной области, при этом пропорции изображения могут быть нарушены. При значении msInside этого свойства изображение помещается внутри выделенной области с сохранением пропорций.

Компонент FontMaster является невизуальным и предназначен для задания единых параметров шрифта для группы компонентов отчета. С его свойством Font в Инспекторе компонентов отчета связано диалоговое окно определения параметров шрифта. Эти параметры шрифта служат для определения параметров шрифта всех компонентов отчета, в свойстве FontMirror которых указан данный компонент FontMaster.

Компонент PageNumInit является невизуальным и предназначен для задания нумерации страниц, начиная с той, где он расположен. Начальный номер при этом определяется свойством InitValue. При необходимости его значение можно загрузить из базы данных с помощью свойств InitDataView и InitDataField.

Компоненты отображения данных

Страница **Report** многостраничной панели инструментов визуального конструктора отчетов содержит компоненты, служащие для представления данных. Их можно разделить на компоненты отображения данных из связанных с ними полей просмотра данных и компоненты управления структурой размещения данных (структурные компоненты), на которые помещаются компоненты отображения данных.

К компонентам *отображения данных* относятся следующие.

- Компонент DataText служит для представления строковых или числовых значений полей связанного с ним просмотра данных.
- Компонент DataMemo обеспечивает представление данных формата Memo или BLOB.
- Компонент CalcText служит для представления результатов вычисления агрегатной функции по значениям связанного поля. Выполняемая функция выбирается из списка возможных значений свойства CalcType.
- Компонент DataMirrorSection является невизуальным и подобно своему потомку Section служит для объединения других компонентов в одну группу.

У всех указанных компонентов представления данных свойство DataView задает связь с просмотром данных, а свойство DataField задает связь с полем просмотра, данные из которого отображаются компонентом.

К *структурным компонентам* относятся следующие.

- Компонент Region служит для выделения области (части страницы отчета), на которой размещаются другие компоненты отображения данных. Более того, компоненты Band и DataBand размещаются только в области, определяемой компонентом Region. Свойство Columns этого компонента определяет число колонок при печати соответствующей части отчета.

- Компонент `Band` задает полосу отчета, на которой располагаются другие компоненты отчета — отображения текста и графики, штрихкоды, отображения данных. Эта полоса служит для оформления заголовков, сносок, верхних и нижних колонтитулов и других фрагментов отчета, не изменяющиеся при печати просмотра данных.
- Компонент `DataBand` задает полосу отчета, представляющую модель строки просмотра данных. На ней размещают компоненты отображения данных. При печати для каждой строки просмотра данных выводится новый экземпляр полосы со всеми расположенными на ней компонентами отчета. Тем самым в отчете построчно отображается весь просмотр данных.

У компонентов `Band` и `DataBand` имеется свойство, которое определяет стиль (назначение и поведение) полосы отчета. С этим свойством в Инспекторе компонентов отчета связан редактор стиля полосы Band Style Editor. В его окне (рис. 23.5) отображается взаимосвязь полос (компонентов `Band` и `DataBand`) в общей области (определяемой компонентом `Region`) и настраивается стиль текущей полосы.

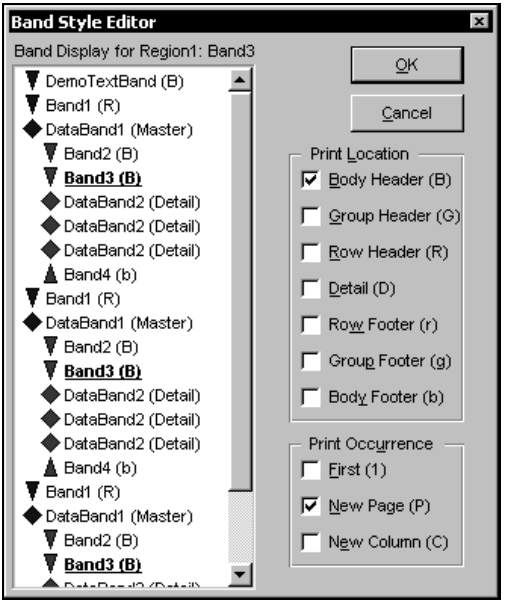


Рис. 23.5. Диалоговое окно **Band Style Editor**

В левой части окна редактора Band Style Editor отображается список полос области со взаимосвязями (реляционные отношения, группировка и вложенность и т. д.). Текущая полоса выделяется полужирным начертанием с подчеркиванием.

Полосы типа `Band` и `DataBand` верхнего уровня (т. е. не являющиеся подчиненными другим полосам типа `DataBand`) помечаются слева цветным ромбиком. При этом подчиненные полосы отчета типов `Band` и `DataBand` определяются с помощью их свойства `ControllerBand`, в качестве значения которого указывается имя полосы типа `DataBand`, которой они подчинены. Подчиненные полосы помечаются слева треугольником, острый угол которого направлен в сторону главной полосы, а цвет совпадает с цветом ромбика главной полосы.

В правой части окна редактора Band Style Editor с помощью группы флажков **Print Location** задается назначение текущей полосы отчета: **Body Header (B)** — заголовок в начале отчета; **Group Header (G)** — заголовок в начале группы, определенной в просмотре с помощью выражения `GROUP BY`; **Row Header (R)** — заголовок в начале каждой записи; **Detail (D)** — заголовок в начале подчиненного набора записей в отношении "один-ко-многим"; **Row Footer (r)** — окончание записи; **Group Footer (g)** — окончание группы записей; **Body Footer (b)** — окончание отчета. С помощью группы флажков **Print Occurrence** задается местоположение полосы в отчете: **First (1)** — на первой странице; **New Page (P)** — на новой странице, **New Column (C)** — в начале каждой колонки отчета.

Компоненты `DataCycle`, `CalcOp`, `CalcTotal` и `CalcController` являются невизуальными и предоставляют ряд дополнительных возможностей по вычислению данных на основе просмотров и отображению их в отчетах.

Компоненты управления отчетом

Компоненты, предназначенные для управления отчетами, находятся на странице **Rave** Палитры компонентов главного окна Delphi. С помощью этих компонентов можно выполнить все действия по настройке параметров, установлению соединения с источником данных, предварительному просмотру и печати имеющегося отчета с использованием генератора отчетов Rave Reports. Другое дело, что ряд этих действий, например предварительный просмотр и печать отчета, удобнее выполнить с помощью визуального конструктора отчетов Rave Reports.

Далее кратко освещены важнейшие компоненты управления отчетом, без которых практически невозможны предварительный просмотр и печать отчетов вне среды визуального конструктора отчетов.

Компонент-проект отчета

При подготовке и использовании отчета обязательным является использование компонента-проекта отчета `RvProject`. С помощью свойства `ProjectFile` типа `String` этого компонента в окне Инспектора компонентов отчета или программно устанавливается связь с проектом отчета (файлом проекта отчета с расширением rav). Например:

```
RvProject1.ProjectFile:='C:\Program Files\Borland\Delphi7\Rave5\ProjectSTR.rav';
```

Для обеспечения связи компонента `RvProject` с компонентом `RvSystem`, предназначенным для управления отчетом, служит свойство `Engine` типа `TRpComponent`. Естественно, перед заданием ссылки на компонент `RvSystem` последний должен быть предварительно помещен в форму приложения. Этот компонент может отсутствовать в форме приложения отчета, в этом случае при запуске отчета открывается диалоговое окно **Output Options**, в котором выбираются вариант дальнейших действий с отчетом (печать, просмотр или помещение в файл) и, при необходимости, формат файла.

В проекте отчета может быть несколько отчетов. Для их указания в компоненте `RvProject` предназначены свойства `ReportName`, `ReportFullName` и `ReportDesc` (соответственно задающие имя, полное имя и описание отчета), имеющие тип `String` и опреде-

ляющие параметры текущего отчета. Для смены в проекте текущего отчета можно использовать функцию `SelectReport(ReportName: String; FullName: boolean): boolean`. Если параметр `FullName` имеет значение `True`, то параметр `ReportName` задает полное имя отчета, в противном случае — имя отчета.

Для получения списка имен отчетов, входящих в состав проекта, служит процедура `GetReportList(ReportList: TString; FullName: boolean)`. При ее вызове имена отчетов помещаются в список строк `ReportList`. При значении `True` параметра `FullName` в названный список помещаются полные имена отчетов, в противном случае — имена отчетов.

Для выполнения текущего отчета предназначены процедуры `Execute` и `ExecuteReport(ReportName: String)`. Во втором случае имя отчета, задаваемое параметром процедуры, должно совпадать с именем отчета, определяемого свойством `ReportName` рассматриваемого нами компонента.

Компонент управления отчетом

Для управления отчетом служит компонент `RvSystem`, который обеспечивает выбор варианта действий с отчетом (печать, просмотр или помещение в файл) и при необходимости выбор формата файла. Как отмечалось, связь этого компонента с проектом отчета реализуется с помощью ссылки из компонента `RvProject` через свойство `Engine`.

Компонент `RvSystem` в своем составе инкапсулирует объекты, которые обеспечивают вывод отчета в один из трех системных приемников: `SystemPrinter` (печать отчета), `SystemPreview` (предварительный просмотр) и `SystemFiler` (вывод в файл).

Перед направлением отчета одному из системных приемников компонент `RvSystem` открывает диалоговое окно **Output Options**, в котором с помощью группы зависимых переключателей **Printer**, **Preview** и **File** осуществляется выбор системного приемника.

Компоненты установления соединения

Для установления соединения с источниками данных служат следующие компоненты страницы **Rave** Палитры компонентов:

- `RvCustomConnection` — соединение с источниками данных, не относящимися к базам данных (текстовые файлы, электронные таблицы и др.);
- `RvDataSetConnection` — соединение с наборами данных приложения баз данных;
- `RvTableConnection` — соединение с компонентом типа `TTable`, устанавливаемое при использовании механизма BDE;
- `RvQueryConnection` — соединение с компонентом типа `TQuery`, устанавливаемое при использовании механизма BDE.

При помещении в форму проекта приложения указанные компоненты оказываются доступными для выбора при создании просмотров **Direct Data View**. При этом предварительно компонент должен быть связан с соответствующим набором данных.

Для важнейшего из перечисленных, т. е. для компонента `RvDataSetConnection` свойство `DataSet` указывает на имя набора данных. Имя компонента `RvDataSetConnection`, зада-

ваемое с помощью его свойства `Name`, используется в отчете для указания имени соединения в просмотрах данных отчета. Напомним, что используемое в просмотре имя соединения можно определить с помощью библиотеки просмотров **Data View Dictionary** как значение свойства `ConnectionName` для интересующего нас компонента `DataView`.

Схема управления отчетом и подсоединения данных

Мы рассмотрели назначение основных компонентов, служащих для управления отчетом и подсоединения данных к отчету. Отметим, что соединение с источниками данных в отчете можно выполнить двумя способами: с помощью драйвера Rave Reports, а также с помощью компонентов приложения доступа к данным и компонентов соединения. Для наглядности на рис. 23.6 приведена обобщенная схема управления отчетом и подсоединения данных к нему.

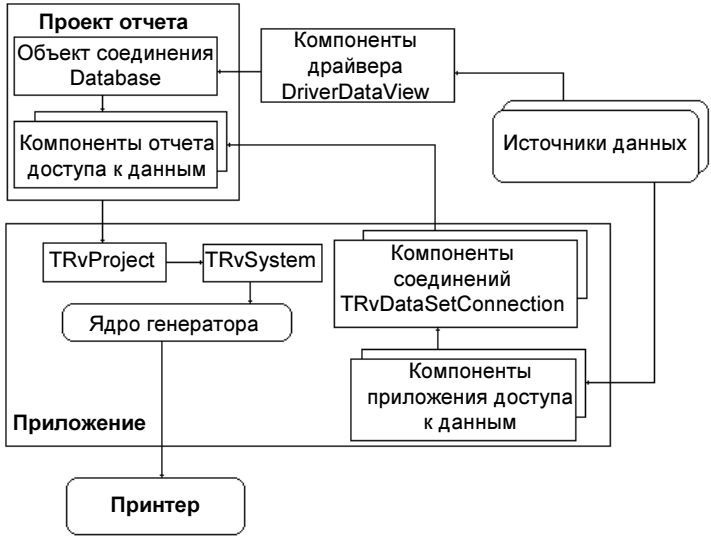


Рис. 23.6. Схема управления отчетом и подсоединения данных

Для организации соединения с помощью драйвера Rave Reports прежде всего командой **File | New Data Object** визуального конструктора отчетов создается объект соединения — компонент `Database`. При этом в открывшемся диалоговом окне **Data Connections** (см. рис. 23.3) выбирается вариант **Database Connection** и в очередном диалоговом окне нужно выбрать тип соединения (ADO, BDE или DBX). Каждому типу соединения соответствует свой драйвер (файл с расширением rdv). Далее в диалоговом окне **Database Connection Parameters** выполняется настройка параметров соединения, состав которых соответствует выбранному типу, и создается объект-соединение (с именем по умолчанию `DadabaseN`).

Теперь для связи компонентов данных отчета с данными соединения требуется создать просмотр данных, выполняющий формирование соответствующего набора данных. Для этого нужно командой **File | New Data Object** визуального конструктора открыть

диалоговое окно **Data Connections** (см. рис. 23.3) и выбрать вариант **Driver Data View**. В очередном диалоговом окне требуется выбрать одно из имеющихся соединений проекта. При этом открывается диалоговое окно **Query Advanced Designer**, в котором на вкладке **Layout** можно указать таблицы из выбранного нами соединения и задать связи между полями таблиц. С помощью вкладки **Sorting & Grouping** окна (рис. 23.7) можно выбрать поля для сортировки и группирования. Кнопка **Editor** открывает окно редактора с текстом сформированного запроса SQL, где его можно откорректировать вручную.

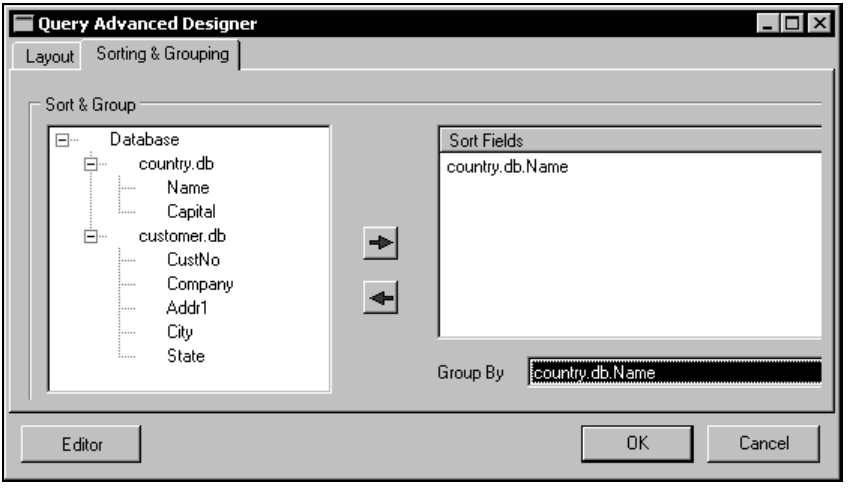


Рис. 23.7. Диалоговое окно **Query Advanced Designer**

После завершения настройки запроса SQL созданный нами объект-просмотр (компонент `DriverDataView`) появляется в словаре **Data View Dictionary** (с именем по умолчанию `DriverDataViewN`). Строка подготовленного таким образом запроса SQL является значением свойства `Query` объекта-просмотра. С его помощью можно повторно вызвать окно редактора Query Advanced Designer.

Теперь созданный нами объект-просмотр данных можно подключать к компонентам доступа к данным в отчете, таким как `DataText`, `DataMemo` и `DataBand`, с помощью их свойства `DataView`. Для таких компонентов, как `DataText` и `DataMemo`, с помощью свойства `DataField` выбирается имя поля из таблицы объекта-просмотра.

Организация соединения с помощью компонентов приложения доступа к данным и компонентов соединения выполняется следующим образом. Для каждого соединения в приложении предварительно создается компонент набора данных, например, `Table` или `Query`, который соединяется с источником данных с помощью его свойств `DatabaseName` и `TableName`. Далее с таким компонентом набора данных устанавливается соединение с помощью соответствующего компонента: `RvCustomConnection`, `RvDataSetConnection`, `RvTableConnection`, `RvQueryConnection` страницы **Rave** Палитры компонентов.

Напомним, что для компонента `RvDataSetConnection` свойство `DataSet` указывает на имя набора данных. Имя компонента `RvDataSetConnection`, задаваемое с помощью его свой-

ства `Name`, используется в отчете для указания имени соединения в просмотрах данных отчета. Как отмечалось, используемое в просмотре имя соединения можно определить с помощью библиотеки просмотров **Data View Dictionary** как значение свойства `ConnectionName` для компонента `DataView`.

Теперь в среде визуального конструктора отчетов требуется создать объект прямого просмотра, с помощью команды **File | New Data Object** открыв диалоговое окно **Data Connections** (см. рис. 23.3) и выбрав вариант **Direct Data View**. Затем в очередном диалоговом окне **Data Connections** нужно выбрать интересующее нас соединение.

Как и при предыдущем способе организации соединения, созданный таким образом объект просмотра данных — компонент `DataView` — можно подключать к компонентам доступа к данным в отчете, например таким, как `DataText`, `DataMemo` и `DataBand`.

Примеры создания и просмотра отчетов

Создание отчета включает три основных этапа:

- создание заготовки отчета в среде визуального конструктора;
- настройка соединения и подключение просмотра данных в случае создания отчета для приложения баз данных;
- помещение в форму приложения и настройка компонентов `RvProject` и `RvSystem`, которые осуществляют связь приложения с заготовкой отчета и управление предварительным просмотром и печатью отчета.

Заготовку собственно отчета в среде визуального конструктора можно выполнить так:

- путем настройки одного из уже имеющихся отчетов, например, в проекте RaveDemo.rav (находится в папке Delphi7\Rave5\Demos);
- путем конструирования отчета для обычного приложения или для приложения баз данных в среде визуального конструктора;
- для приложения баз данных можно создать простой табличный отчет или отчет для отношений типа "главный — подчиненный" с помощью Мастера отчетов.

В приведенном списке вариантов создания заготовки отчета второй вариант представляется наиболее трудоемким и требует хороших навыков работы с компонентами визуального конструктора.

Кроме перечисленных выше действий, при создании отчета можно также выполнять преобразование файлов отчета в различные форматы и др.

Предварительный просмотр отчета

Чтобы создать отчет путем настройки одного из уже имеющихся в проекте RaveDemo.rav (находится в папке Delphi7\Rave5\Demos) отчетов, нужно выполнить предварительный просмотр представляющего интерес отчета, а затем уже заняться его настройкой.

Рассмотрим, как выполнить предварительный просмотр отчета `TwoDetails` из проекта RaveDemo.rav. Для отчетов приложений баз данных, использующих просмотры данных, к которым относится отчет `TwoDetails`, перед просмотром и выполнением нужно настроить соединения. Вид отчета `TwoDetails` в окне визуального конструктора на этапе проектирования и соответствующее ему дерево отчета приведены на рис. 23.8.

Чтобы определить имена просмотров данных, используемых компонентами отображения данных отчета (`DataText1`, `DataText2` и `DataText3`), поочередно выделим эти компоненты в окне проекта отчета и с помощью Инспектора компонентов отчета выясним значение свойства `DataView`. В нашем случае это просмотры с именами `CustomerDV`, `OrdersDV` и `ItemsDV`.

Далее нам нужно определить используемые этими просмотрами имена соединений. Для этого в словаре просмотров данных **Data View Dictionary** также поочередно выделим просмотры с полученными именами и по значению их свойств `ConnectionName` выясним имена соединений. В нашем случае это значения `CustomerCXN`, `OrdersCXN` и `ItemsCXN` соответственно.

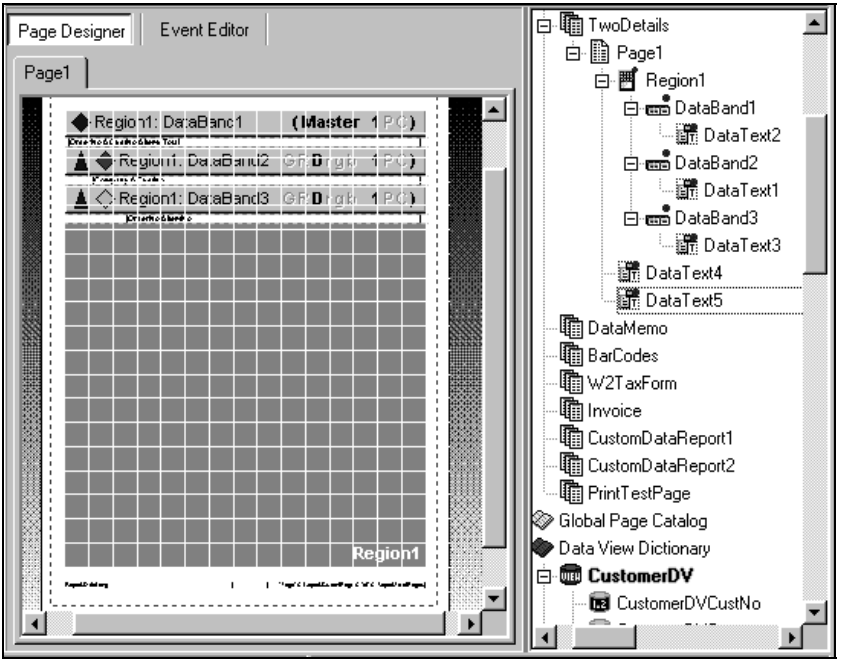


Рис. 23.8. Вид отчета `TwoDetails` в окне визуального конструктора

Все это означает, что в приложении баз данных имеются три набора данных, с которыми мы должны установить связь с помощью трех компонентов `RvDataSetConnection` (размещенных на странице **Rave** Палитры компонентов) с именами `CustomerCXN`, `OrdersCXN` и `ItemsCXN`. Для связи компонентов `RvDataSetConnection` с наборами данных из базы данных используется свойство `DataSet` этих компонентов. В качестве наборов данных в нашем примере используются таблицы из базы данных `DBDEMOS` с именами `orders.db`, `customer.db` и `items.db`.

После настройки соединения отчета с базой данных приложения можно выполнить предварительный просмотр отчета `TwoDetails` в среде визуального конструктора, задав для этого команду **File | Execute Report**. Выберем в открывшемся диалоговом окне **Output Options** вариант **Preview** и нажмем кнопку **OK**. Вид отчета `TwoDetails` в окне предварительного просмотра приведен на рис. 23.9.

Из вида отчета на рис. 23.9 и из значений свойства `DataField` компонентов отображения данных `DataText1`, `DataText2` и `DataText3` видно, что в отчете компонент `DataText1` отображает данные из трех полей `OrderNO`, `CustomNO` и `ItemsTotal` таблицы `orders.db`; компонент `DataText2` отображает данные из полей `Company` и `CustNO` таблицы `customer.db`; компонент `DataText3` отображает данные из полей `OrderNO` и `ItemNO` таблицы `items.db`. Кроме того, компоненты `DataText4` и `DataText5` соответственно отображают номер текущей страницы из общего числа страниц и дату просмотра отчета.

Полученная нами при предварительном просмотре информация, а также данные о составе компонентов и их свойствах позволяют при необходимости настроить отчет соответствующим образом. Например, можно изменить таблицы базы данных приложения, можно изменить имена отображаемых полей и т. п.

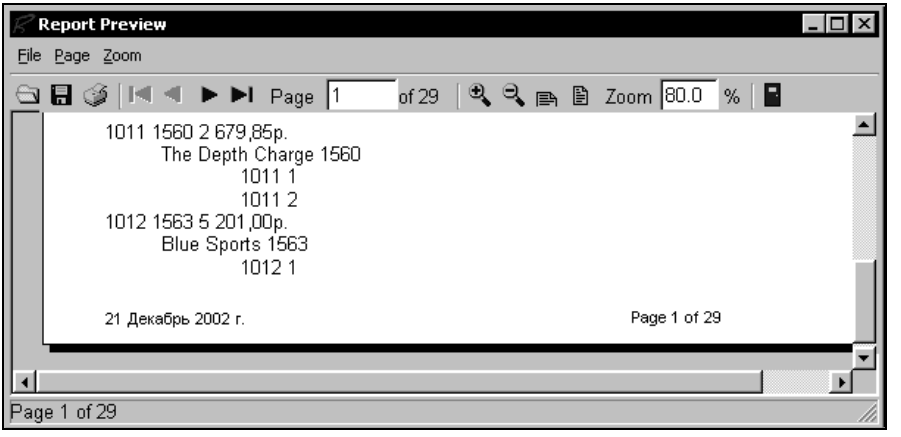


Рис. 23.9. Вид отчета `TwoDetails` при предварительном просмотре

Простой отчет приложения базы данных

Простой отчет представляет собой отчет на основе данных из одного набора данных и содержит сведения, которые выводятся в табличном виде без какой-либо дополнительной обработки данных (например, группирования). Размещение и вид печатаемых в отчете данных аналогичны размещению и виду данных, отображаемых в сетке `DBGrid`. Отличием является то, что данные отчета не размещаются в форме, а представлены в виде бумажного документа, и их нельзя редактировать.

Рассмотрим последовательность шагов, которую требуется выполнить при добавлении простого отчета к приложению базы данных.

1. Откроем приложение базы данных Delphi, для которого требуется добавить простой отчет.

2. В форму приложения базы данных со страницы **Rave** Палитры поместим компонент `RvDataSetConnection` и с помощью Инспектора объектов присвоим свойству `DataSet` этого компонента значение, указывающее на имя набора данных, используемого в приложении.
3. С помощью визуального конструктора отчетов подготовим отчет и создадим файл проекта отчета. Для этого:
   - с помощью команды **Tools | Rave Designer** меню Delphi запустим визуальный конструктор Rave Reports 5.0;
   - выбором команды **File | New Data Object** откроем диалоговое окно **Data Connections** (см. рис. 23.3) и выберем в нем вариант **Direct Data View**;
   - в очередном диалоговом окне в списке **Active Data Connections** выберем вариант **RVDataSetConnection1** и нажмем кнопку **Finish**;
   - в дереве проекта в правой части окна визуального конструктора отчетов раскроем узел **Data View Dictionary** и в нем раскроем вновь созданный узел **DataView1**;
   - выбрав команду **Tools | Report Wizards | Simple Table**, запустим Мастер создания простых таблиц в отчете, выберем вариант **DataView1** и нажмем кнопку **Next**;
   - на последующих шагах работы с мастером выберем поля таблицы для отображения в отчете, при необходимости изменим очередность следования полей, установим параметры полей страницы, текст заголовков и шрифтов, используемых в отчете (рис. 23.10);

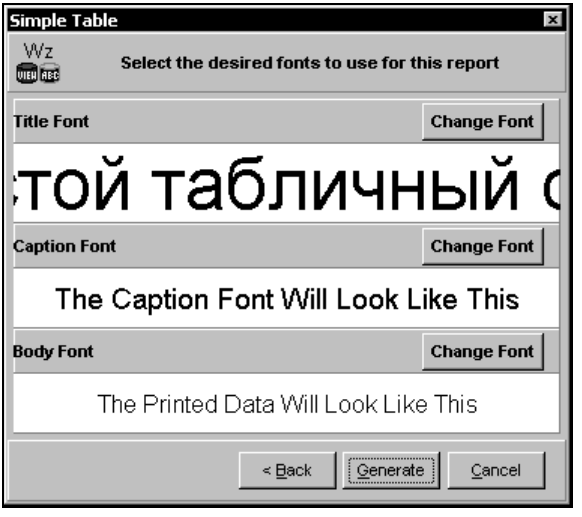


Рис. 23.10. Диалоговое окно настройки параметров шрифта

   - на заключительном этапе работы с мастером нажатием кнопки **Generate** запустим процесс генерации отчета;

- для просмотра сгенерированного отчета выберем команду **File | Execute Report**, в открывшемся диалоговом окне **Output Options** в поле **Report Destination** выберем переключатель **Preview** и нажмем **OK**. Вид полученного нами отчета при просмотре приведен на рис. 23.11;

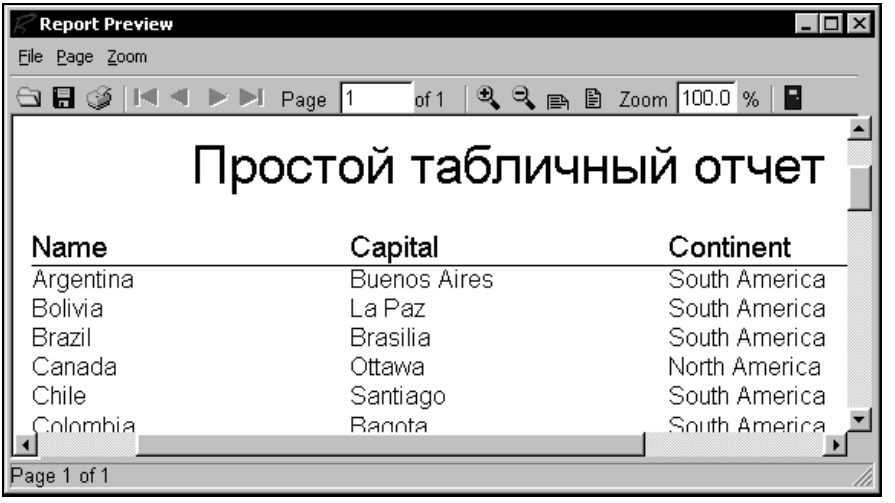


Рис. 23.11. Вид простого табличного отчета при просмотре

- при необходимости выполним настройку параметров (шрифта, цвета и др.) отдельных составляющих созданного отчета;
- сохраним созданный нами проект отчета в файле с произвольным именем, например STRep, и расширением rav с помощью команды **File | Save As**;
- свернем или закроем диалоговое окно работы с визуальным конструктором отчета и вернемся к работе с приложением в Delphi.

4. В форму приложения базы данных со страницы **Rave** Палитры поместим компонент проекта отчета `RvProject` и установим в значение `C:\Program Files\Borland\Delphi7\Rave5\STRep.RAV` (спецификация созданного нами файла проекта) свойство `ProjectFile` этого компонента с помощью Инспектора объектов.

5. В форму приложения базы данных со страницы **Standard** Палитры компонентов поместим кнопку (компонент `Button`) и в качестве обработчика события `OnClick` нажатия этой кнопки зададим вызов метода `ExecuteReport`, обеспечивающего выполнение отчета с заданным именем (в нашем примере `Report5`) из состава проекта отчета (компонента `RvProject1`), например, так:

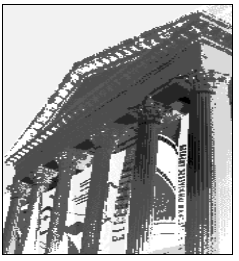
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RvProject1.Open;
  try
    RvProject1.ExecuteReport('Report5');
```

```
    finally
      RvProject1.Close;
    end;
  end;
```

6. С помощью клавиши <F9> запустим приложение на выполнение и в форме щелкнем мышью на кнопке, за которой закреплен созданный нами обработчик события. В открывшемся диалоговом окне выберем вариант печати отчета и нажмем кнопку **OK**.

В результате выполненных действий будет выдан на печать простой табличный отчет, содержащий данные из таблицы приложения базы данных, вид которого в окне просмотра приведен на рис. 23.11.

ГЛАВА 24



Инструменты

Инструменты — это программы, предназначенные для обслуживания БД, а также для выполнения вспомогательных действий при разработке приложений, например, для создания таблиц и отладки SQL-запросов. Совместно с Delphi поставляется большое число инструментов, применимых для работы как с локальными, так и с удаленными БД. В этой главе рассматриваются инструменты для работы с локальными базами данных. Инструментальные программы, предназначенные для работы с удаленными БД, представлены в *главе 28*.

Программа BDE Administrator

Программа BDE Administrator представляет собой инструмент администрирования процессора баз данных BDE (далее Администратор BDE, или Администратор). Для вызова Администратора BDE запускается файл bdeadmin.exe, находящийся в одном каталоге с процессором баз данных. Внесенные изменения сохраняются по окончании работы с Администратором в файле конфигурации idapi32.cfg. Программу также можно вызвать через главное меню Windows выбором пункта **Program | Borland Delphi 7 | BDE Administrator**.

Администратор BDE позволяет настраивать параметры БД и операционной системы. Основные параметры:

- параметры псевдонима БД:
  - название;
  - тип;
  - путь;
- параметры драйвера:
  - тип;
  - язык;

- системные установки:
  - установки по умолчанию;
  - форматы даты, времени и числовые форматы.

Для настройки требуемого параметра в левой части окна Администратора BDE нужно выбрать необходимый объект, после чего в правой части окна открывается доступ к списку параметров этого объекта. При редактировании выбранного в панели инструментов объекта становятся доступными кнопки отмены/подтверждения сделанных изменений (с красной/синей стрелкой). Отменить или подтвердить изменения также можно командами **Cancel** и **Apply** главного или контекстного меню. Слева от объектов, имеющих неутвержденные изменения, отображается зеленый треугольник.

Добавить новый объект можно, выбрав в окне Администратора пункт меню **Object | New**. Удаление выделенного объекта выполняется командой **Object | Delete** главного или контекстного меню или нажатием кнопки **Delete** панели инструментов (с синим косым крестом).

Текущее состояние объекта, выбранного в левой части окна, отображает значок, который появляется слева от имени объекта. Варианты значков и их связь с состоянием объекта перечислены далее:

- зеленый треугольник — объект находится в режиме редактирования (имеет неутвержденные изменения);
- зеленый треугольник с красными лучами — вновь созданный и еще не сохраненный в конфигурации объект, для которого выполняется редактирование;
- красный треугольник — объект находится в режиме редактирования, некоторые изменения являются некорректными и не могут быть сохранены;
- красный треугольник с красными лучами — созданный и еще не сохраненный в конфигурации объект находится в режиме редактирования, некоторые данные являются некорректными и не могут быть сохранены;
- ярко-зеленый квадрат — объект открыт.

Из приложения можно управлять настройками BDE с помощью соответствующих методов компонента `Session`, который рассматривается в *главе 25*.

Работа с псевдонимами

Псевдоним (alias) указывает местонахождение файлов БД и представляет собой специальное имя для обозначения каталога. Использование псевдонимов существенно облегчает перенос файлов БД в другие каталоги и на другие компьютеры. При этом не требуется изменять приложение, которое осуществляет доступ к таблицам БД. Если в приложении расположение таблиц указано с помощью псевдонима, то после перемещения БД для обеспечения работоспособности приложения достаточно изменить значение пути, заданное в псевдониме. Если же в приложении путь к БД указан в явном виде, т. е. без псевдонима, то после перемещения БД нужно изменять само приложение — вносить изменения в исходный код и заново его транслировать.

Для создания псевдонима базы данных перед вызовом пункта меню **Object | New** Администратора BDE нужно выбрать вкладку **Database** в левой части окна, в противном

случае команда **New** меню будет недоступна. После задания этой команды появляется диалоговое окно **New Database Alias** (Новый псевдоним БД), в котором нужно выбрать тип драйвера. Для локальных таблиц Paradox и dBase выбирается тип `STANDARD`, для других таблиц указывается соответствующий тип, например, для удаленного сервера InterBase — тип `INTRBASE`.

После нажатия кнопки **OK** создается псевдоним, и его данные отображаются в окне Администратора BDE (рис. 24.1). Новый псевдоним автоматически получает имя `STANDARD1` и параметры по умолчанию. Можно переименовать псевдоним, выполнив команду **Rename** контекстного меню псевдонима или меню **Object** главного меню Администратора BDE.

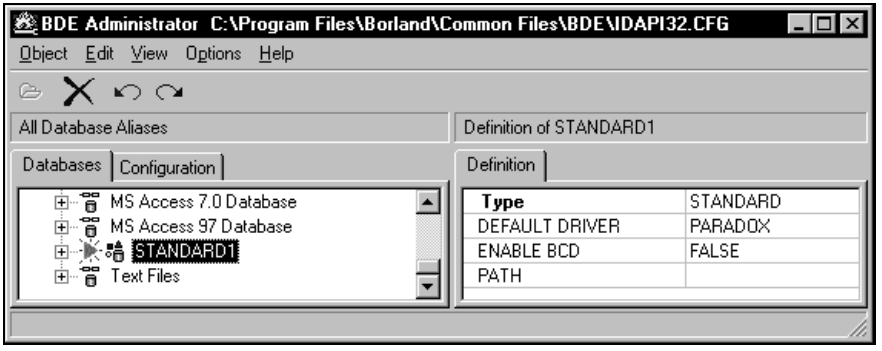


Рис. 24.1. Установка параметров псевдонима

Псевдоним для работы с локальными БД имеет три параметра.

- `DEFAULT DRIVER` — указывает формат таблиц БД (по умолчанию имеет значение `Paradox`). Кроме того, можно установить значения `dBase` или `ASCIIDRV` для текстовых файлов, разбитых на колонки.
- `ENABLE BCD` — указывает на необходимость перевода чисел в формат BCD, что позволяет более точно выполнять вычисления, но уменьшает скорость их выполнения. По умолчанию имеет значение `False` и, соответственно, формат BCD не используется.
- `PATH` — указывает расположение (каталог) БД. После создания псевдонима путь не определен, и разработчик должен установить его самостоятельно.

Отметим, что псевдонимы для удаленных БД имеют большее число параметров, например, у псевдонима типа `INTRBASE` пятнадцать параметров.

При необходимости можно изменить параметры псевдонима, например, имя и путь. Для параметров `DEFAULT DRIVER` и `ENABLE BCD` значение выбирается в раскрывающемся списке. Значение параметра `PATH` можно ввести вручную или с помощью выбора нужного каталога в окне **Select Directory**, которое появляется при двойном щелчке в поле значения параметра. После выбора диска, каталога и нажатия кнопки **OK** соответствующий путь автоматически присваивается параметру `PATH` в качестве значения.

Смена пути выполняется при перемещении БД в другой каталог.

Ненужный псевдоним можно удалить, выполнив команду **Delete** контекстного меню псевдонима или команду **Object | Delete** главного меню Администратора.

Для каждого псевдонима указывается соответствующий драйвер, для локальных таблиц — обычно `dBase` или `Paradox`, при этом параметры драйвера устанавливаются по умолчанию.

Для псевдонима типа `INTRBASE`, предназначенного для доступа к удаленной БД InterBase, в меню **Object** программы появляется команда **Diagnostics**. Эта команда открывает окно **Communication Diagnostic Tool** (Диагностирование соединения) проверки соединения с удаленной БД (рис. 24.2).

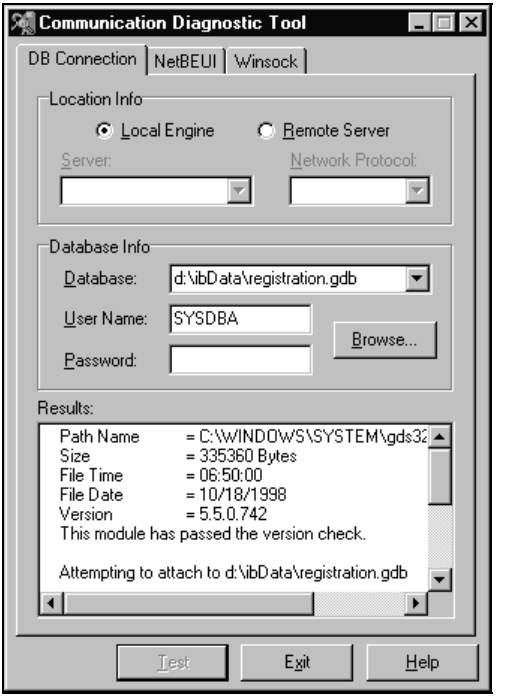


Рис. 24.2. Диалоговое окно проверки соединения с удаленной БД

После указания параметров соединения (расположения БД, имени и пароля пользователя) и нажатия кнопки **Test** выполняется соединение с БД, результаты которого выводятся в поле **Results**. Более подробно вопросы, касающиеся соединения с удаленной БД, рассматриваются в *главе 25*.

Для работы с псевдонимами можно использовать и другие программы, например, Database Desktop.

Параметры драйвера

Для доступа к параметрам драйвера в левой части окна Администратора нужно выбрать вкладку **Configuration** и требуемый драйвер. Драйверы процессора баз данных делятся на "родные" (**Native**) для него драйверы и драйверы ODBC. При выборе соот-

ветствующего драйвера в правой части окна появляется список параметров драйвера, которые можно просматривать или изменять. В качестве примера на рис. 24.3 показаны параметры локального драйвера базы данных Paradox.

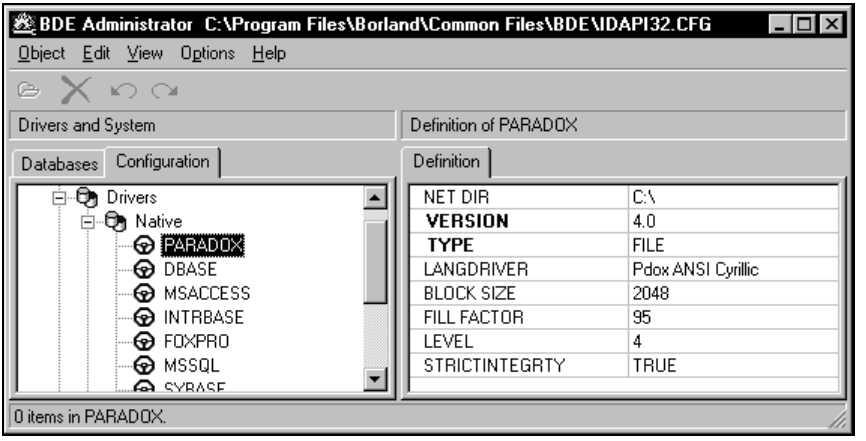


Рис. 24.3. Установка параметров драйвера базы данных Paradox

Наибольший интерес представляют параметры `TYPE` и `LANGDRIVER`.

Параметр `TYPE` указывает тип драйвера и принимает следующие значения:

- `FILE` — для локальных БД;
- `SERVER` — для удаленных БД.

Параметр `LANGDRIVER` определяет драйвер языка (языковой драйвер), используемый для кодировки символов. В нашей стране для этого параметра рекомендуется использовать значения `dBASE RUS cp866` и `Pdox ANSI Cyrillic` соответственно для драйверов dBase и Paradox. Это обеспечивает корректное отображение символов кириллицы в приложениях, их правильную сортировку и преобразование, например, при использовании функций `AnsiUpperCase` и `AnsiLowerCase`.

Замечание

Параметры драйвера действуют только на те приложения, которые осуществляют доступ к БД через процессор баз данных.

Как отмечалось, у драйвера базы данных InterBase гораздо больше параметров (рис. 24.4), чем у драйвера Paradox.

Наибольший интерес представляют параметры, перечисленные далее.

- `DLL32` — указывает драйвер SQL-Links, используемый для доступа к БД (sqlint32.dll).
- `ENABLE BCD` — указывает на необходимость перевода чисел в формат BCD, что позволяет более точно выполнять вычисления, но уменьшает скорость их выполнения; по умолчанию имеет значение `False`, и формат BCD не используется.
- `LANGDRIVER` — определяет драйвер языка, используемый для кодировки символов; выбор языкового драйвера `Pdox ANSI Cyrillic` обеспечивает корректную работу с символами русского алфавита.

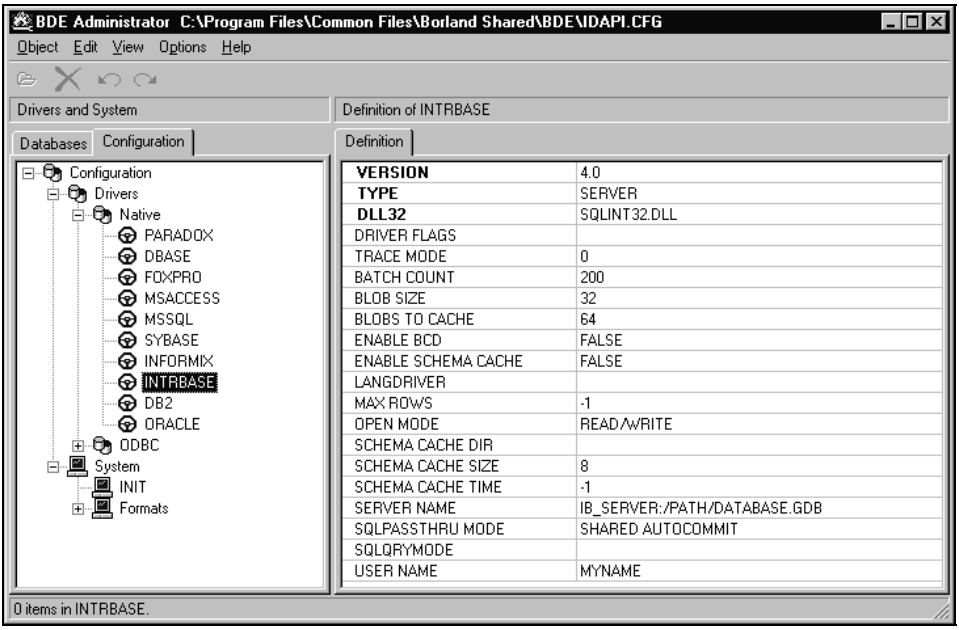


Рис. 24.4. Установка параметров драйвера InterBase

- `MAXROWS` — определяет максимальное число записей, которые могут быть считаны из удаленной БД при одном запросе к ней. Этот параметр используется для блокировки попыток считывания большого объема информации при ошибочном или неправильно сформированном SQL-запросе к удаленной БД. Установка небольшого значения параметра `MAXROWS` часто приводит к ошибкам, связанным с тем, что возвращаемое на основании запроса число записей может легко превысить установленное ограничение. По умолчанию параметр имеет значение `-1`, что соответствует отсутствию ограничений на число возвращаемых записей.
- `OPEN MODE` — определяет режим доступа к данным. По умолчанию имеет значение `READ/WRITE`, что обеспечивает как чтение, так и изменение записей. При установке значения `READ ONLY` разрешается только чтение записей.
- `SERVER NAME` — указывает имя удаленной БД. Формат задания этого имени зависит от сетевого протокола. Например, для протокола TCP/IP имя БД состоит из имени сервера, пути к БД и собственно имени БД.
- `SQLPASSTHRU MODE` — определяет способ взаимодействия процессора баз данных BDE с сервером на уровне транзакций. По умолчанию имеет значение `SHARED AUTOCOMMIT`, при котором в случае, если приложение явно не управляет транзакциями, сервер выполняет это управление автоматически.
- `SQLQRYMODE` — задает режим выполнения запросов. Параметр имеет следующие возможные значения:
  - `LOCAL` — запрос выполняется локально (на компьютере пользовательского приложения);
  - `SERVER` — запрос выполняется на сервере; если сервер не может выполнить запрос, то данный запрос не выполняется;

- если не выбрано ни одно из этих значений (по умолчанию), запрос посылается серверу; если сервер не может выполнить запрос, то он выполняется локально.

♦ `USER NAME` — задает начальное имя пользователя, которое при соединении с БД содержится в соответствующем поле. По умолчанию имеет значение `MYNAME`.

Для доступа к удаленным БД с помощью BDE используются драйверы SQL-Links:

♦ Microsoft Access — idda3532.dll, iddao32.dll;

♦ Microsoft SQL Server — sqlmss32.dll;

♦ SyBase — sqlssc32.dll, sqlsyb32.dll;

♦ Informix — sqlinf9.dll, sqlinf32.dll;

♦ InterBase — sqlint32.dll;

♦ DB2 — sqldb2v5.dll, sqldb232.dll;

♦ Oracle — sqlora32.dll, sqlora8.dll.

Эти драйверы поставляются совместно с процессором баз данных BDE и находятся в его каталоге.

Если для БД нет драйвера SQL Links, то для него можно использовать драйвер ODBC. Отметим, что ODBC (Open DataBase Connectivity — совместимость открытых баз данных) представляет собой интерфейс прикладного программирования (API) в виде библиотеки функций, вызываемых из различных программных сред и позволяющих приложениям унифицированно обращаться на языке SQL к базам данных различных форматов. В рамках концепции ODBC есть стандарт для драйверов. Разработчики многих БД для доступа к ним предоставляют драйверы, соответствующие этому стандарту, обеспечивая тем самым совместимость различных типов баз.

Настройка драйверов ODBC выполняется с помощью программы-администратора ODBC, окно **ODBC Data Source Administrator** которого открывается через элемент **ODBC Data Source (32-bit)** Панели управления Windows. Администратор ODBC можно вызвать также из Администратора BDE командой **ODBC Administrator** контекстного меню драйвера.

Отметим, что для работы с БД могут существовать драйверы как SQL-Links, так и ODBC. В этом случае предпочтительнее использовать драйверы SQL-Links, т. к. они работают быстрее.

Системные установки

Системные установки делятся на два вида: установки по умолчанию и форматы.

Параметры системных *установок по умолчанию* становятся доступными при выборе в окне Администратора BDE объекта **Configuration\System\INIT**. Основными параметрами этих установок являются следующие:

♦ `DATA REPOSITORY` (активный словарь данных);

♦ `DEFAULT DRIVER` (драйвер, устанавливаемый для локальных БД), по умолчанию драйвер Paradox; устанавливается как драйвер по умолчанию для каждой новой БД;

- LANGDRIVER (драйвер языка); рекомендуется установить этот параметр в значение dBASE RUS cp866 (для драйвера dBase) или Pdox ANSI Cyrillic (для драйвера Paradox);
- LOCAL SHARE (флаг разделяемого доступа к локальным БД между приложениями); по умолчанию имеет значение False, и разделяемый доступ к локальным данным запрещен;
- SQLQRYMODE (режим выполнения запросов).

Системные установки форматов позволяют задавать следующие параметры форматов даты, времени и чисел:

- параметры даты:
  - SEPARATOR (разделитель дня, месяца и года); по умолчанию применяется символ из системных установок Windows;
  - MODE (порядок расположения дня (Д), месяца (М) и года (Г) в дате); принимает следующие значения: 0 (МДГ), 1 (ДМГ), 2 (ГМД); по умолчанию используется порядок расположения, заданный в системных установках Windows;
  - FOURDIGITYEAR (количество цифр, отображаемых для года), принимает значения: False (две цифры) — по умолчанию, True (четыре цифры);
  - YEARBIASED (способ преобразования значения года, заданного двумя цифрами), принимает значения: True (к двузначному числу прибавляется число 2000) — по умолчанию, False (значение года не изменяется);
  - LEADINGZEROM (незначащий ноль для месяца, обозначенного одной цифрой), принимает значения: True (добавляется незначащий ноль) — по умолчанию, False (ноль не добавляется);
  - LEADINGZEROD (незначащий ноль для дня, обозначенного одной цифрой), принимает значения: True (добавляется незначащий ноль) — по умолчанию, False (ноль не добавляется);
- параметры времени:
  - TWELVEHOUR (формат времени), принимает значения: True (время отображается в 24-часовом формате) — по умолчанию, False (время отображается в 12-часовом формате);
  - AMSTRING (символы для 12-часового формата, обозначающие время до полудня), по умолчанию используются символы AM;
  - PMSTRING (символы для 12-часового формата, обозначающие время после полудня), по умолчанию используются символы PM;
  - SECONDS (отображение секунд), принимает значения: True (секунды отображаются) — по умолчанию, False (секунды не отображаются);
  - MILLISECONDS (отображение миллисекунд), принимает значения: True (миллисекунды отображаются), False (миллисекунды не отображаются) — по умолчанию;

- числовые параметры:
  - DECIMALSEPARATOR (разделитель целой и дробной части числа); по умолчанию используется символ из системных установок Windows;
  - THOUSENDSEPARATOR (разделитель тысяч в целой части числа); по умолчанию используется символ из системных установок Windows;
  - DECIMALDIGIT (число разрядов в дробной части числа) — по умолчанию имеет значение 2;
  - LEADINGZERON (незначащий ноль для чисел, по модулю меньших единицы), принимает значения: True (ноль отображается), False (ноль не отображается) — по умолчанию.

Системные установки Windows можно изменять через Панель управления.

Использование конфигурационных файлов

Как отмечалось ранее, по окончании работы с Администратором BDE и после сохранения внесенных изменений параметры запоминаются в файле конфигурации idapi32.cfg. Этот конфигурационный файл используется процессором баз данных автоматически (по умолчанию).

Кроме того, имеется возможность сохранить выполненные настройки в собственном конфигурационном файле. Для этого нужно выполнить команду главного меню Администратора BDE **Object | Save As Configuration**. В появившемся окне указываются каталог и имя конфигурационного файла (cfg). После нажатия кнопки **Save** указанный файл сохраняется на диске и доступен для последующего использования.

Для загрузки (открытия) собственного конфигурационного файла нужно выполнить команду **Object | Open Configuration**. После выбора конфигурационного файла и нажатия кнопки **Open** содержащиеся в нем установки вступают в действие.

Можно объединить несколько конфигурационных файлов. Для этого нужно выбрать команду **Object | Merge Configuration**. После выбора файла и нажатия кнопки **Open** данные открытого и выбранного конфигурационных файлов объединяются и соответствующие им установки вступают в действие.

Этой возможностью удобно пользоваться при наличии нескольких приложений, обращающихся к процессору баз данных. Процессор баз данных можно включить только в дистрибутив приложения, устанавливаемого первым. При установке последующих приложений кроме файлов приложения желательно иметь и конфигурационные файлы этих приложений. После объединения данных файлов с конфигурационным файлом idapi32.cfg необходимые настройки, например, псевдонимы приложений, становятся доступными процессору баз данных.

Отметим, что при настройке компьютера пользователя вместо объединения конфигураций можно ввести необходимые данные вручную. Однако это более трудоемкий вариант, он рекомендуется только когда таких данных немного.

Программа Database Desktop

Программа Database Desktop предназначена для создания и редактирования таблиц, визуальных запросов и SQL-запросов, а также для выполнения действий с псевдонимами БД. Эту программу можно вызвать из среды Delphi командой **Tools | Database Desktop** или запуском файла dbd32.exe, находящегося в одном каталоге с файлами программы Database Desktop (по умолчанию это каталог \Database Desktop). Программу также можно вызвать через главное меню Windows, выбрав команду **Program | Borland Delphi 7 | Database Desktop**.

Создание и изменение структуры таблиц уже были рассмотрены в *главе 16* при описании технологии создания приложения БД. Здесь мы познакомимся с применением программы Database Desktop для выполнения остальных действий.

Если при манипулировании таблицами и запросами не указывается расположение БД, то по умолчанию они считаются расположенными в рабочем каталоге программы Database Desktop. Для изменения этого каталога нужно выполнить команду **File | Working Directory** меню программы, в результате чего появится окно установки каталога **Set Working Directory** (рис. 24.5).

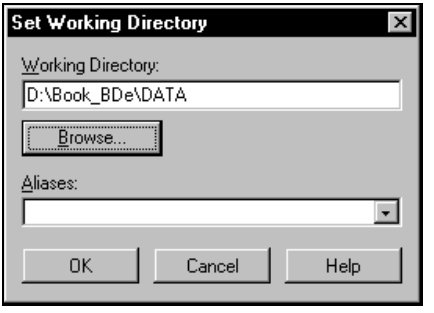


Рис. 24.5. Установка рабочего каталога

Задать рабочий каталог можно двумя способами:

- непосредственно ввести путь в поле **Working Directory** или нажатием кнопки **Browse** открыть окно просмотра каталогов **Directory Browser**, в котором выбрать нужные диск (псевдоним) и каталог;
- выбрать в списке псевдонимов **Aliases** ранее определенный псевдоним.

При одновременном использовании обоих способов рабочий каталог выбирается по полю **Working Directory**. После задания рабочего каталога и нажатия кнопки **OK** выбранные установки вступают в действие.

Сразу после инсталляции программа Database Desktop неправильно отображает символы русского алфавита, поэтому нужно изменить используемый ею шрифт. Для этого командой **Edit | Preferences** нужно открыть окно настройки **Preferences**, в поле **Default system font** которого выводится название используемого по умолчанию шрифта. Нажатие кнопки **Change** открывает окно выбора шрифта. В этом окне следует выбрать но-

вый шрифт, содержащий символы русского алфавита, например, Arial или Times New Roman.

Редактирование записей таблиц

Чтобы открыть таблицу, нужно выбрать команду **File | Open | Table** или нажать на панели инструментов кнопку с изображением таблицы. Затем в открывшемся окне **Open Table** нужно выбрать имя главного файла таблицы. После нажатия кнопки **Open** в поле программы Database Desktop открывается окно с записями таблицы.

По умолчанию для таблицы включен режим просмотра данных **View Data**, и ее записи доступны только для чтения. Переключение в режим редактирования **Edit Data** выполняется одноименной командой меню **Table**. После этого в таблице можно добавлять и удалять записи, а также изменять значения полей.

Добавление записи выполняется нажатием клавиши <Insert>, новая запись появляется над текущей записью, в любом поле которой установлен курсор ввода. Удаляется текущая запись комбинацией клавиш <Ctrl>+<Delete>.

Для редактирования значения поля нужно установить в него курсор ввода (мышью или клавишами управления курсором) и нажать любую алфавитно-цифровую клавишу, при этом старое значение поля удаляется. Фиксация нового значения выполняется нажатием клавиши <Enter> или переводом курсора ввода в другое поле или другую запись. Перейти к редактированию поля можно также двойным щелчком мыши, при этом старое значение поля не удаляется. Возврат в режим просмотра таблицы выполняется командой **Table | View Data**.

Режим просмотра и редактирования включается с помощью кнопок панели инструментов, на которых изображены буквы **ab** и карандаш на фоне таблицы соответственно. При включении режима кнопка остается нажатой. Переключение между режимами выполняется с помощью клавиши <F9>.

Работа с псевдонимами

Для работы с псевдонимами БД используется программный инструмент Alias Manager (Менеджер псевдонимов, рис. 24.6), вызываемый командой **Tools | Alias Manager** меню программы Database Desktop.

С помощью Менеджера псевдонимов можно создавать (кнопка **New**) и удалять (кнопка **Remove**) псевдонимы. Имя псевдонима вводится (для нового) или выбирается (для существующего) в списке **Database alias**. Кроме того, можно изменять параметры псевдонимов: тип драйвера (список **Driver type**) и путь (поле **Path**). Путь можно ввести вручную или выбрать в окне просмотра каталогов, открываемом нажатием кнопки **Browse**.

Нажатие кнопки **Save As** открывает окно сохранения конфигурационного файла (idapi.cfg) процессора баз данных. При необходимости можно запомнить конфигурацию под другим именем.

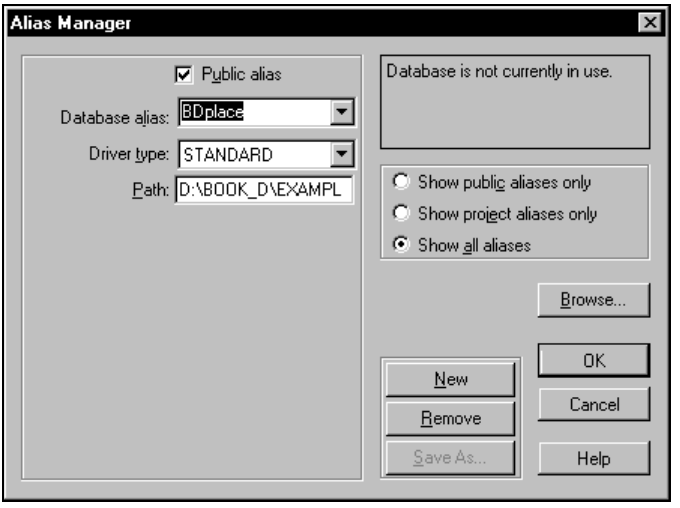


Рис. 24.6. Диалоговое окно Менеджера псевдонимов

Работа с SQL-запросами

Работа с SQL-запросами включает:

- создание запроса;
- редактирование запроса;
- выполнение запроса.

Для создания SQL-запроса нужно, выполнив команду **New | SQL File** программы Database Desktop, открыть окно редактора SQL Editor (рис. 24.7) и набрать в нем текст SQL-запроса.

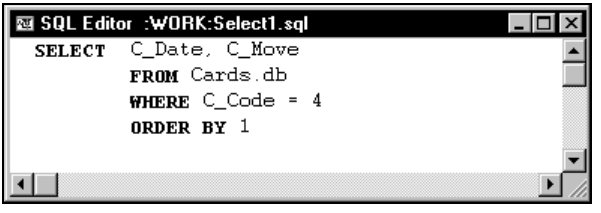


Рис. 24.7. Окно редактора SQL-запроса

Как и любой другой документ, запрос можно сохранять (**Save**) на диске в виде текстового файла, сохранять под другим именем (**Save As**), а также редактировать (в окне SQL Editor). Можно также открыть ранее сохраненный запрос (**Open**). Файл запроса имеет расширение sql. (В скобках указаны соответствующие команды пункта **File** программы Database Desktop.)

Для выполнения SQL-запроса нужно выбрать команду **SQL | Run SQL** меню программы Database Desktop или нажать кнопку  панели инструментов. Перед выполнением запроса производится проверка правильности его синтаксиса.

Отметим, что для работы с локальными таблицами в программе Database Desktop реализована версия SQL-92, несколько отличающаяся от стандарта. В частности, в ней больше, чем в стандарте, типов полей и упрощены конструкции отдельных операторов.

При выполнении правильно оформленного запроса появляется окно с записями, отобранными в результате обработки запроса (рис. 24.8).

Рис. 24.8. Результат выполнения SQL-запроса

В среде программы Database Desktop удобно отлаживать SQL-запросы, которые в последующем можно присвоить в качестве значения свойству `SQL` компонента `Query`.

Визуальное конструирование запросов

Для удобного конструирования запросов можно использовать визуальный конструктор, вызываемый командой **New | QBE Query** программы Database Desktop. При этом открывается окно **Select File**, где нужно указать имя главного файла таблицы, на основании данных которой строится запрос. После выбора файла и нажатия кнопки **Open** открывается окно визуального конструктора (рис. 24.9).

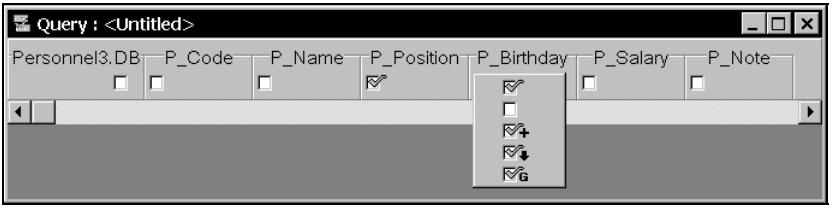


Рис. 24.9. Диалоговое окно визуального конструктора запросов SQL

В этом окне показаны имена таблицы и всех ее полей. Под именем каждого поля находятся флажок и текстовое поле (слева от флажка для имени таблицы и справа — для имени поля).

С помощью визуального конструктора можно:

- создавать и изменять запрос по образцу;
- выполнять запрос по образцу;
- сохранять запрос по образцу как SQL-запрос.

Бóльшая часть работы с запросом по образцу проходит в окне визуального конструктора, где для каждого поля таблицы задаются условия отбора и сортировки. Для этого

щелчком правой кнопки мыши в поле имени таблицы (слева от флажка) вызывается контекстное меню, в котором выбирается вид запроса:

- пустая строка (отбор и редактирование записей таблицы);
- `INSERT` (вставка записей в таблицу);
- `DELETE` (удаление записей из таблицы);
- `SET` (сравнение записей в таблицах).

Для выполнения запроса нужно выполнить команду **Query | Run Query** программы Database Desktop или нажать кнопку  панели инструментов. Перед выполнением запроса производится проверка его правильности.

Для получения текста запроса, соответствующего визуальному запросу, нужно выполнить команду **Query | Show SQL** или нажать на панели инструментов кнопку с буквами **SQL**. При отсутствии ошибок в конструкции запроса автоматически открывается окно **SQL Editor** (Редактор SQL-запроса, см. рис. 24.7), в котором содержится текст запроса на языке SQL. При наличии ошибок окно Редактора SQL-запроса не появляется, а выдается сообщение об ошибке.

Запрос можно сохранять (**Save**) на диске под своим именем или под другим (**Save As**). Можно также открыть ранее сохраненный запрос (**Open**). Файл запроса имеет расширение qbe. (В скобках указаны команды меню **File** программы Database Desktop.)

Отбор записей из таблицы

При отборе записей из таблицы щелчком правой кнопки мыши на флажках полей вызывается контекстное меню (см. рис. 24.9), в котором можно выбрать метку для флажка:

- пусто (поле не включается в результат запроса);
- галочка со знаком + (поле включается в результат запроса);
- галочка (поле включается в результат запроса с сортировкой записей по возрастанию значений этого поля);
- галочка с черной стрелкой (поле включается в результат запроса с сортировкой записей по убыванию значений этого поля);
- галочка с буквой **G** (поле включается в результат запроса и используется для группирования записей).

Кроме того, справа от флажка для каждого поля таблицы можно вручную ввести условие отбора записей. Поле, для которого допускается ввод условия запроса, обозначается текстовым курсором. Этот курсор отображается черным прямоугольником и переключается мышью или клавишами управления курсором. При вводе (редактировании) условия отбора черный прямоугольник сменяется мигающей вертикальной линией.

При выполнении запроса, показанного на рис. 24.10, из таблицы `Personnel` отбираются записи, для которых значение оклада (поле `P_Salary`) больше или равно `3500`. Кроме поля оклада, в результат запроса включаются поля имени (`P_Name`) и должности (`P_Position`).

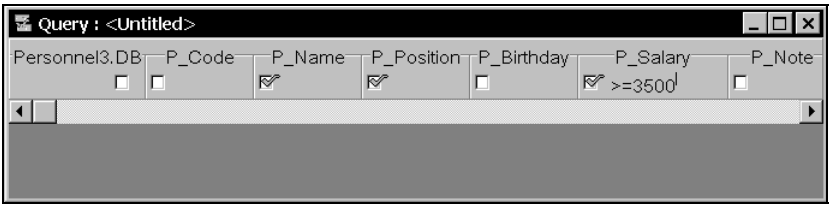


Рис. 24.10. Отбор записей по величине оклада

В приведенном примере поле `P_Salary` используется для отбора записей и одновременно включается в результат запроса. В общем случае включать поле, для которого задано условие отбора записей, в результат запроса не обязательно. Таким образом, записи можно отбирать на основании одних полей, а в результат запроса включать другие.

При выполнении запроса, показанного на рис. 24.11, в набор данных, как и в предыдущем примере, отбираются записи, для которых значение поля `P_Salary` больше или равно `3500`. Но в результат запроса включается только поле `Name`.

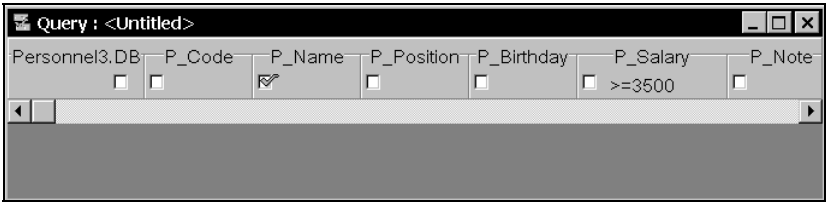


Рис. 24.11. Отбор записей по полю, не входящему в результат запроса

Условие отбора может быть более сложным, чем просто сравнение, и содержать операции логического умножения (`AND`) и сложения (`OR`).

Для объединения отдельных условий с помощью операции логического умножения нужно перечислить эти условия в одной строке через запятую. Например, при выполнении запроса, показанного на рис. 24.12, будут отобраны записи, для которых значение поля `P_Salary` находится в интервале `3500...5000`.

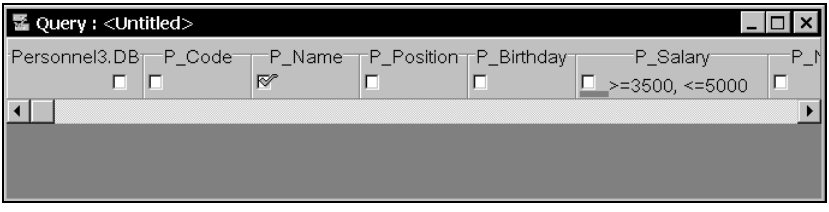


Рис. 24.12. Использование в условии отбора операции логического умножения

Для соединения отдельных условий операцией логического сложения нужно перечислить эти условия в разных строках. Так, при выполнении запроса, показанного на рис. 24.13, в набор попадут записи, для которых поле `P_Position` содержит значения `Водитель` или `Секретарь`. В результат запроса не включаются поля кода и примечания.

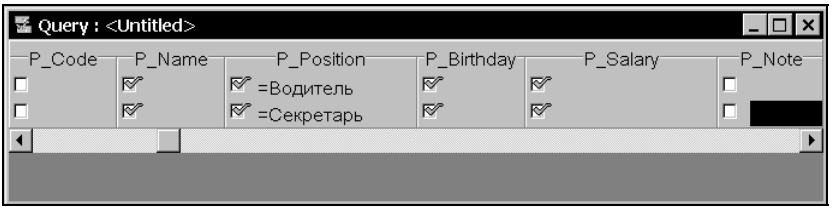


Рис. 24.13. Использование в условии отбора операции логического сложения

Добавление к запросу новой строки выполняется переводом текстового курсора вниз при нахождении его на последней строке (с помощью клавиш управления курсором). При этом курсор может находиться в любом поле запроса.

Замечание

Отметки для каждого поля во всех строках запроса должны быть одинаковыми. Например, если в приведенном на рис. 24.13 запросе во второй строке убрать отметку для поля `P_Birthday`, то при попытке выполнить запрос будет выдано сообщение об ошибке.

Удаление строки запроса выполняется одновременным нажатием клавиш <Ctrl> и <Delete>. Предварительно текстовый курсор должен быть установлен в любом поле удаляемой строки.

При отборе записей по нескольким полям условие отбора задается отдельно для каждого поля. Условия связываются операцией логического умножения — таким образом, в результат запроса попадают записи, для полей которых выполняются все условия.

Например, при выполнении запроса, показанного на рис. 24.14, будут отобраны записи, для которых значение поля `P_Position` равно `Водитель`, а значение поля `P_Salary` больше или равно `3000`. В результат запроса попадают поля `P_Name`, `P_Position` и `P_Salary`.

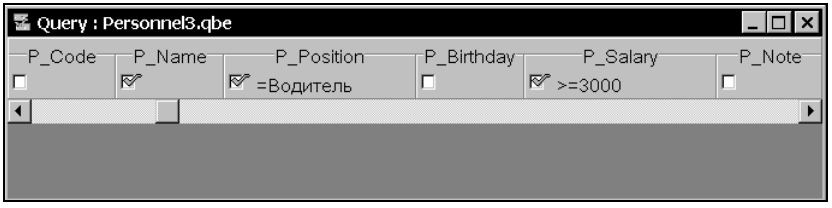


Рис. 24.14. Отбор записей по нескольким полям

Редактирование записей

При редактировании записей необходимо определить:

- условия отбора записей;
- новые значения изменяемых полей.

Условия отбора записей формируются, как рассмотрено в предыдущем разделе. Для изменяемых полей следует задать новые значения, перед которыми указывается ключевое слово `CHANGETO`.

Замечание

При вставке и удалении записей флажки всех полей должны быть пустыми.

При выполнении запроса, показанного на рис. 24.15, для всех записей, имеющих значение `Секретарь` поля `P_Position`, значение поля `P_Salary` будет установлено в `4700`.

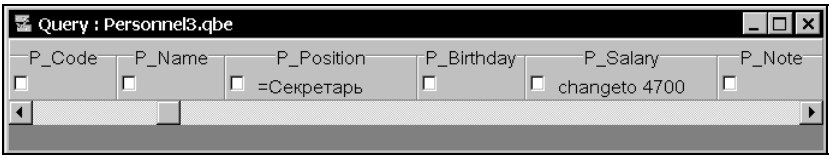


Рис. 24.15. Редактирование записей

Аналогично, в одном запросе можно отобрать записи на основании более сложного условия, а также одновременно изменить значения нескольких полей.

Вставка и удаление записей

При вставке и удалении записей справа от флажков для полей таблицы указываются значения соответствующих полей. При вставке записей нужно с помощью контекстного меню в поле имени таблицы выбрать в качестве типа запроса `INSERT`.

При выполнении запроса, показанного на рис. 24.16, в таблицу `Personnel3` добавляется новая запись, имеющая следующие значения полей: `P_Name` — `Семенова Е.Н.`, `P_Position` — `Менеджер`, `P_Birthday` — `12.03.67`, `P_Salary` — `4000`.

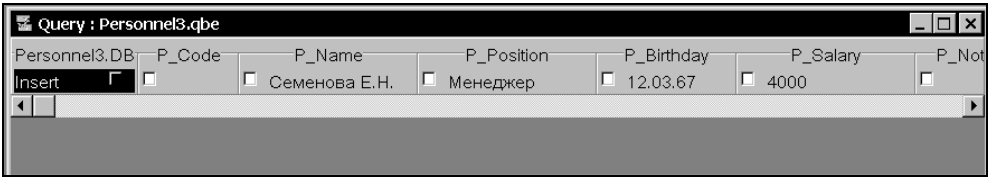


Рис. 24.16. Вставка записи

Замечание

При вставке и удалении записей флажки всех полей должны быть пустыми.

Значения отдельных полей можно не задавать, однако если поле требует обязательного ввода, то при выполнении запроса возникнет ошибка.

В результате отработки одного запроса в таблицу можно добавить сразу несколько записей. При этом значения полей каждой новой записи вводятся отдельной строкой запроса. Например, при выполнении запроса, показанного на рис. 24.17, в таблицу `Personnel3` добавляются две записи, значения полей первой: `P_Name` — `Семенова Е.Н.`, `P_Position` — `Менеджер`, `P_Birthday` — `12.03.67`, `P_Salary` — `4000`, а второй: `P_Name` — `Петров В.Э.`, `P_Salary` — `4500`, `P_Position` — `Менеджер`. Поля `P_Code` обеих записей явля-

ются автоинкрементными и заполняются автоматически. Поле `P_Birthday` второй записи и поля `P_Note` обеих записей остаются пустыми.

Рис. 24.17. Одновременная вставка двух записей

Замечание

При вставке нескольких записей вид запроса `INSERT` в поле имени таблицы нужно указывать для каждой записи.

При удалении записей с помощью контекстного меню в поле имени таблицы следует выбрать вид запроса `DELETE`. Справа от флажков полей указываются значения полей удаляемых записей.

Например, запрос, показанный на рис. 24.18, удаляет запись, имеющую значения полей: `P_Name` — `Семенова E.H.`, `P_Position` — `Менеджер`, `P_BirthDay` — `12.03.67`, `P_Salary` — `4000`.

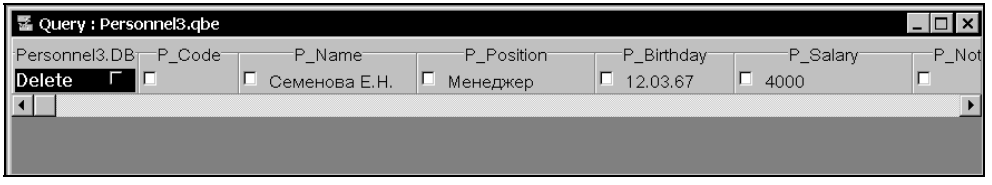


Рис. 24.18. Удаление записи

При выполнении запроса на удаление записей возможны следующие ситуации:

- в таблице нет записи с указанными значениями полей — удаление записи не происходит;
- в таблице есть несколько записей с указанными значениями полей — удаляются все такие записи.

Если значение поля не задано, то это поле не влияет на отбор записей для удаления. В запросе, показанном на рис. 24.19, такими полями являются, например, `P_Name` и `P_Birthday`. В результате его выполнения из таблицы `Personnel3` удаляются все записи, соответствующие менеджерам с окладом 4000.

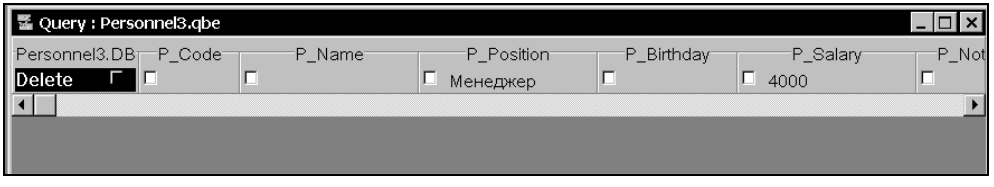


Рис. 24.19. Удаление записи

Можно задать более сложные критерии отбора записей для удаления, например, определив вторую строку условия (рис. 24.20). При выполнении этого запроса из таблицы `Personnel3` удалятся все записи, соответствующие менеджерам и водителям с окладом 4000.

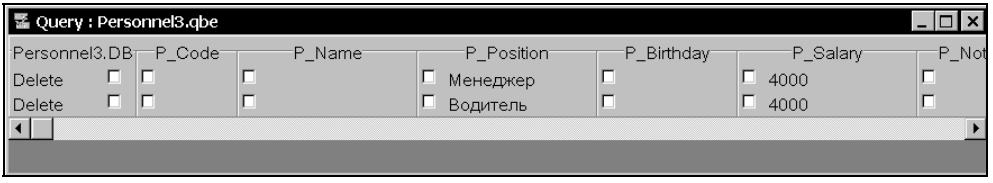


Рис. 24.20. Удаление записей

В качестве результата запроса, выполняющего добавление или удаление записей, возвращается совокупность добавленных или удаленных записей, соответствующих заданным в запросе условиям.

Связывание таблиц

Можно построить запрос по образцу не только для одиночных таблиц, как это рассмотрено выше, но и для связанных таблиц. Для этого требуется:

- добавить к запросу новую таблицу;
- связать между собой две таблицы.

Добавление таблицы выполняется нажатием кнопки **Add Table** панели инструментов (на кнопке изображены таблица и знак +) и выбором в открывшемся окне **Select File** главного файла таблицы.

Для связывания таблицы нужно нажать кнопку **Join Tables** панели инструментов (на кнопке изображены две таблицы), при этом кнопка останется в нажатом состоянии, а к указателю мыши добавляется изображение двух таблиц. Затем щелчком на полях связи обеих таблиц между ними устанавливается соединение, получающее имя `join1`. Имя соединения отображается в полях связи. Так же можно связать между собой три и более таблиц.

Для связанных таблиц операции отбора, редактирования, вставки и удаления записей выполняются так же, как и для одиночных таблиц. Рассмотрим в качестве примера технику отбора записей. Запрос, показанный на рис. 24.21, выводит из подчиненной таблицы `Cards` записи с движением товара, который указан в главной таблице `Store` склада. Отбор записей осуществляется по полю `S_Name` названия товара, в результирующий набор данных включаются поля `C_Move` количества поступившего или убывшего товара и `C_Date` даты выполнения операции. При выполнении запроса будут выведены записи, соответствующие покупке и продаже моркови.

Если требуется, например, вывести записи за определенный период, то для поля `C_Date` дополнительно указываются нижнее и верхнее значения даты.

Для удаления таблицы из запроса нужно нажать кнопку **Remove Tables** панели инструментов (с изображением таблицы и знака –), в открывшемся одноименном окне выбрать имя таблицы и нажать кнопку **OK**.

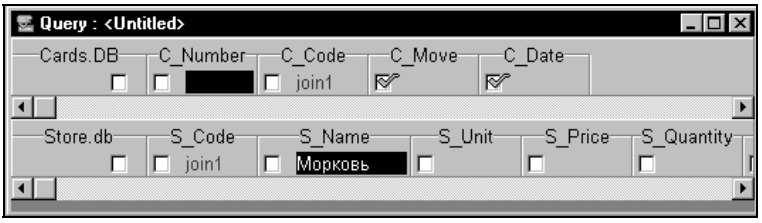


Рис. 24.21. Связывание таблиц в программе Database Desktop

Замечание

После удаления одной из таблиц в поле связи другой таблицы остается имя связи.

Для удаления связи нужно поочередно установить курсор в поля связи обеих таблиц, где отображается имя связи, например, `join1`, и нажать клавишу <Delete>.

Программа SQL Builder

Программа SQL Builder предназначена для упрощения создания SQL-запросов. С помощью этой программы разработчику достаточно удобно конструировать запросы, которые сохраняются в виде текстового файла с расширением sql.

Программа SQL Builder (рис. 24.22) вызывается выбором команды **SQL Builder** контекстного меню компонента `Query`. Отметим, что во время работы программы SQL Builder нельзя перейти в другие окна Delphi, такой переход возможен только по завершении работы с ней.

В верхней части окна программы SQL Builder размещаются таблицы, выбранные для построения запроса, а в нижней части — многостраничный блокнот.

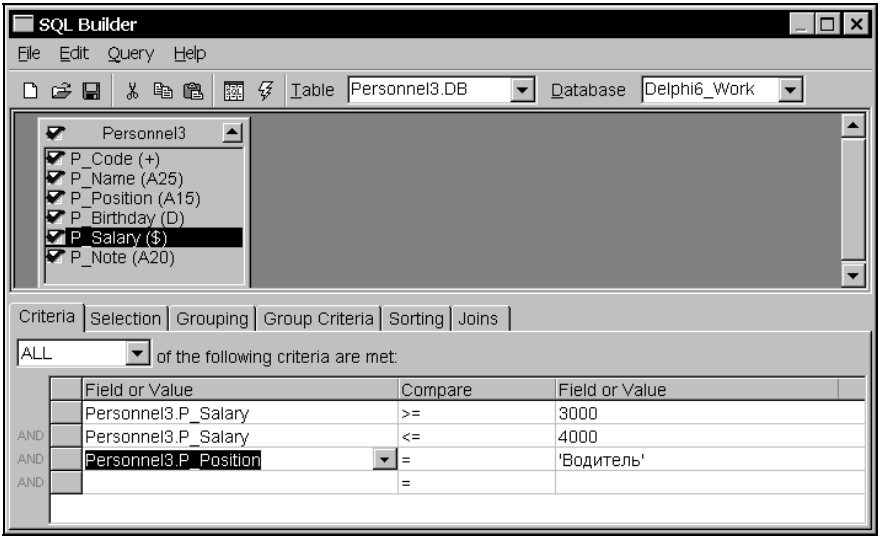


Рис. 24.22. Диалоговое окно программы SQL Builder

Для выполнения запроса нужно вызвать команду **Query | Run Query** программы SQL Builder или нажать кнопку панели инструментов. Перед выполнением запроса производится проверка его правильности. На рис. 24.23 показано окно с результатами выполнения запроса.

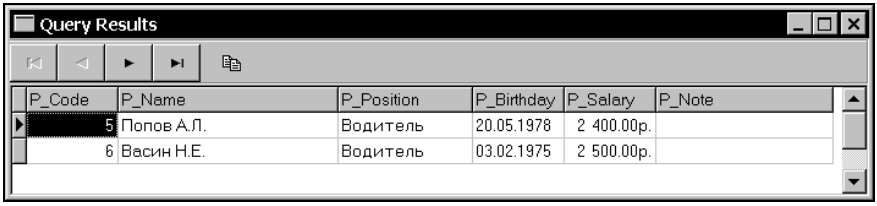


Рис. 24.23. Диалоговое окно с результатами выполнения запроса

Для получения текста, соответствующего визуальному запросу, нужно выполнить команду **Query | Show SQL** или нажать на панели инструментов кнопку с буквами **SQL**. При отсутствии ошибок в конструкции запроса автоматически открывается окно **SQL Query Text Entry** (рис. 24.24), в котором содержится текст запроса на языке SQL. В этом окне можно выполнить такие действия, как сохранение, загрузка или выполнение запроса.

При наличии ошибок названное окно не появляется, а выдается сообщение об ошибке.

Сохранение запроса в виде текстового файла с расширением sql выполняется командой **File | Export to file**. Открытие запроса осуществляется командой **File | Import from file**.

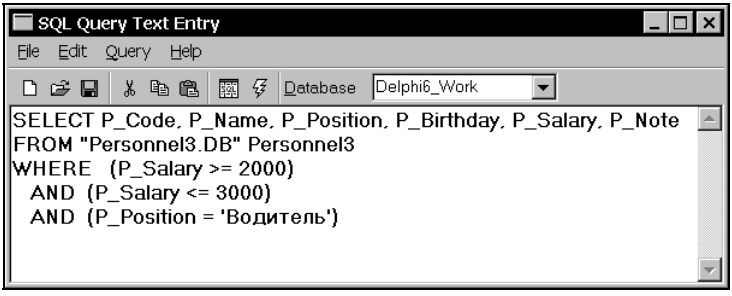


Рис. 24.24. Окно **SQL Query Text Entry**

Замечание

Если сохраненный запрос содержит ошибки, то при попытке его открыть выдается соответствующее сообщение, и запрос не открывается. Поэтому перед сохранением запроса в sql-файле следует проверять его правильность, например, путем его выполнения.

Сохранить и открыть запрос можно также с помощью кнопок панели инструментов, на которых изображены дискета и открытая папка соответственно. Отметим, что всплывающие подсказки этих кнопок не совпадают с названиями соответствующих команд меню. Для кнопки сохранения — это **Save the current query** (Сохранить текущий запрос), а для кнопки открытия — **Open an existing SQL Query** (Открыть существующий SQL-запрос).

Чтобы добавить в окно программы новую таблицу, нужно в поле списка **Database** указать расположение базы данных, выбрав псевдоним в раскрывающемся списке или введя путь к каталогу с файлами БД вручную. После этого в списке **Table** выбирается нужная таблица, которая автоматически добавляется в верхнюю часть окна. Добавляемые таблицы могут располагаться и в разных каталогах.

Таблица представляется изображением, похожим на комбинированный список. В верхней части этого списка находятся псевдоним (alias) таблицы, флажок и стрелка. Стрелка позволяет свернуть или развернуть список, а флажок служит для управления включением всех полей таблицы в результат запроса или выключением из него.

Замечание

Псевдоним таблицы представляет собой имя, используемое в тексте SQL-запроса для замены имени таблицы, и не имеет ничего общего с псевдонимом, указывающим расположение БД и конкретной таблицы.

Для смены псевдонима таблицы нужно вызвать команду **Edit Table Alias** контекстного меню этой таблицы. При этом в псевдониме отображается текстовый курсор, что показывает его готовность к изменению. После нажатия клавиши <Enter> новый псевдоним вступает в действие.

Остальную часть списка занимают строки с информацией о полях таблицы — имя поля, его тип и размер. Размер указывается только для тех полей, для которых он не устанавливается автоматически на основании типа поля, например, для поля строкового типа. Слева от названия поля находится флажок, управляющий включением этого поля в результат запроса. Если флажок установлен (стоит галочка), то поле включается в результат запроса.

Удаление таблицы из окна программы выполняется командой **Remove Table** контекстного меню этой таблицы. Можно также выделить таблицу, щелкнув на ее изображении, при этом вокруг ее псевдонима отобразится пунктирный прямоугольник. После выделения таблица удаляется нажатием клавиши <Delete>.

Программа SQL Builder позволяет достаточно просто и удобно выполнить связывание (соединение) таблиц. Для этого нужно выбрать мышью поле в одной таблице и перетащить его в используемое для связи поле другой таблицы. После отпускания кнопки мыши между таблицами устанавливается связь, что отображается линией, соединяющей эти таблицы (рис. 24.25). В обеих таблицах поля, используемые для связи, отображаются в списке полей первыми (верхними) и выделяются жирным шрифтом. В отличие от ряда других программ, например, Microsoft Access, эта линия соединяет названия таблиц, а не используемые для связи поля обеих таблиц.

Напомним, что в главной таблице поле связи должно быть ключевым, а в подчиненной таблице — индексным. Поля связи должны иметь одинаковые или совместимые типы, в противном случае выдается сообщение об ошибке несоответствия типов.

Для удаления связи нужно щелкнуть правой кнопкой мыши на линии, обозначающей связь между таблицами, вызвав контекстное меню, и выполнить его единственную команду **Delete Join**. После подтверждения этой операции связь удаляется.

В нижней части окна программы SQL Builder находится блокнот, который автоматически появляется при добавлении к окну программы первой таблицы. С помощью блок-

нота можно включать в запрос условия отбора записей или указывать направления их сортировки.

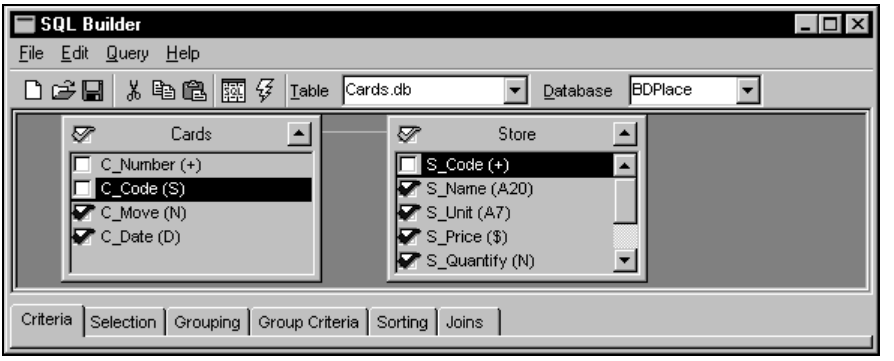


Рис. 24.25. Связывание таблиц

Страница **Criteria** позволяет задать условия отбора записей. Каждое условие вводится в отдельной строке и состоит из двух имен полей или значений, разделенных операцией сравнения. Имена полей выбираются в списке или перетаскиваются мышью из соответствующей таблицы в верхней части окна. Операция сравнения выбирается в списке. Если условие введено с ошибкой, то выдается соответствующее сообщение, а само условие в запрос не включается и отображается красным цветом.

Удаление строки с условием выполняется командой **Delete Row** контекстного меню удаляемой строки. Отметим, что аналогично можно удалить строку и из списка других страниц блокнота.

Список **of the following criteria are met** (из перечисленных условий выполнены) позволяет задать логические операции, которые связывают строки с отдельными условиями. В левой части строк с условиями отображаются названия логических операций. По умолчанию список содержит значение **ALL** (ВСЕ), что соответствует операции логического умножения (см. рис. 24.22) и означает, что все перечисленные условия выполнены. При этом в левой части строк с условиями, кроме первой, отображается операция AND. Кроме значения **ALL**, в списке можно также выбрать:

- **ANY** (операция логического сложения OR);
- **NONE** (инверсия значения **ANY**);
- **NOT ALL** (инверсия значения **ALL**).

Страница **Selection** позволяет изменить названия заголовков столбцов, используемых для отображения значений полей в результирующем наборе, а также включить в запрос статистические функции по полям.

В правой части списка указываются поля, включенные в результат запроса — им соответствуют установленные флажки в изображениях таблиц. В левой части списка для каждого поля помещается название соответствующего столбца, которое по умолчанию совпадает с именем поля (без указания имени таблицы).

В запросе, показанном на рис. 24.26, английские названия столбцов трех первых полей заменены на русские.

Результирующий набор данных (с русскими названиями столбцов) можно видеть на рис. 24.27.

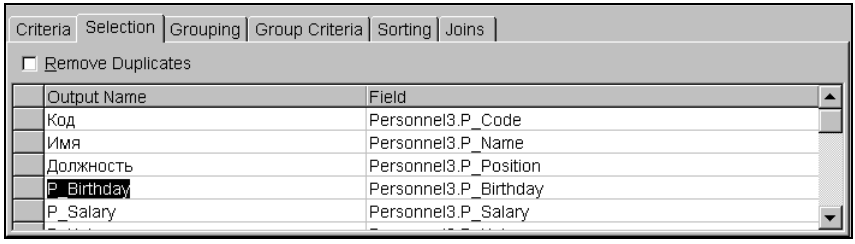


Рис. 24.26. Задание названий для столбцов результирующего набора данных

Рис. 24.27. Вид результирующего набора данных с измененными названиями столбцов

Добавление к запросу статистической функции выполняется с помощью команды **Summary** (Итог) контекстного меню страницы. В результате появляются списки, в которых можно выбрать функцию (например, `SUM`) и поле.

Флажок **Remove Duplicates** (Исключить повторы) позволяет убрать из результирующего набора одинаковые записи. По умолчанию он снят, и в результат попадают все записи (в том числе и совпадающие), которые отвечают заданным условиям отбора.

Страница **Grouping** позволяет сгруппировать записи по полям, при этом список **Output Fields** содержит все поля, которые можно использовать для группирования записей. В список **Grouped On** отбираются имена полей, по которым выполняется группирование (рис. 24.28). Перемещение полей первого списка во второй и удаление полей из второго списка выполняются с помощью кнопок **Add** и **Remove**, которые становятся активными при выборе поля (полей) в соответствующем списке. Переместить поле между списками можно также двойным щелчком на его имени.

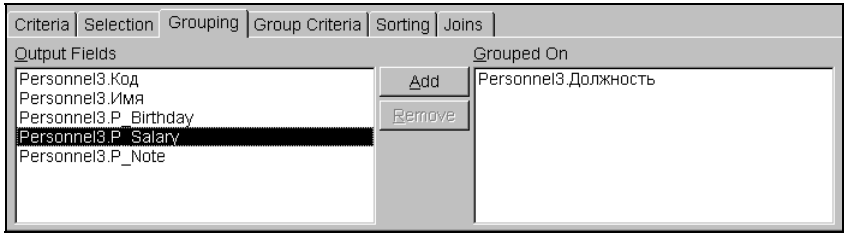


Рис. 24.28. Задание полей для группирования записей

Доступными являются поля, флажок которых в изображении таблицы установлен. Имя поля состоит из псевдонима (имени) таблицы и названия столбца для этого поля, заданного на странице **Selection**.

Страница **Group Criteria** содержит условия, используемые для группирования записей. Условия группирования вводятся аналогично условиям отбора записей, задаваемым на странице **Criteria**.

Страница **Sorting** определяет порядок сортировки записей. Список **Sorted By** содержит имена полей, по которым выполняется сортировка (рис. 24.29). Значение **Sorted Order** (Порядок следования полей) определяет последовательность сортировки: сначала записи упорядочиваются по значениям поля, указанного в списке первым (верхним), затем записи, имеющие одинаковое значение первого поля, упорядочиваются по второму полю и т. д. Для изменения порядка расположения полей, по которым выполняется сортировка, предназначены кнопки в виде черных треугольников, расположенные над именами полей. Кнопка с треугольником, направленным вниз, перемещает выделенное поле (поля) на одну строку вниз, кнопка с треугольником, направленным вверх, — на одну строку вверх.

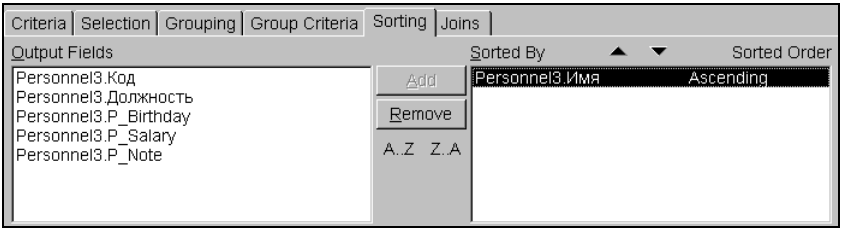


Рис. 24.29. Задание полей для сортировки записей

По умолчанию сортировка выполняется в порядке возрастания значения поля, на что указывает признак **Ascending** справа от имени поля. Для задания обратного порядка сортировки по полю нужно его выделить и нажать кнопку **Z..A**, при этом вместо **Ascending** устанавливается признак **Descending**. Нажатие кнопки **A..Z** возвращает прежний порядок сортировки.

Список **Output Fields** содержит поля, которые можно использовать для сортировки записей. Обозначение полей и способы их перемещения между списками не отличаются от обозначения полей и способов их перемещения для страницы **Grouping**.

Страница **Joins** позволяет устанавливать связи (соединения) между таблицами (рис. 24.30).

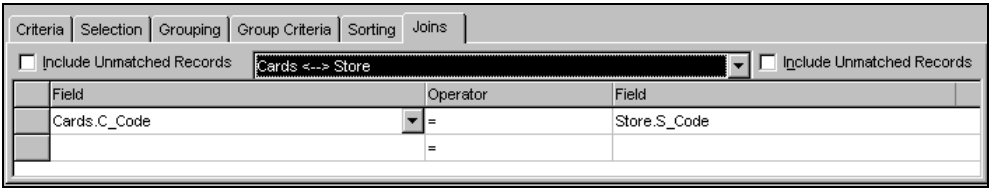


Рис. 24.30. Задание связи между таблицами

Действия, выполняемые на этой странице, частично дублируют описанные ранее действия по связыванию таблиц и разрыве связи между ними. Кроме того, здесь можно задать дополнительные условия для связи и выбрать оператор для сравнения значений полей связываемых таблиц, например, >, < или = (по умолчанию).

Программа SQL Explorer

Программа SQL Explorer представляет собой аналог Проводника Windows, с помощью которого можно просматривать и редактировать базу данных. В зависимости от версии эта программа имеет различные возможности и даже разные названия, например, SQL Explorer, Explorer или Database Explorer. Для запуска SQL Explorer нужно выполнить команду **Database | Explorer** меню Delphi или запустить файл dbexplor.exe, находящийся в каталоге BIN главного каталога Delphi. Программу также можно вызвать через главное меню Windows выбором пункта **Programs | Borland Delphi 7 | SQL Explorer**.

Интерфейс программы SQL Explorer практически не отличается от интерфейса программы BDE Administrator. В левой части окна (рис. 24.31) в древовидной структуре выбирается объект, после чего свойства этого объекта становятся доступными для просмотра и редактирования в правой части окна.

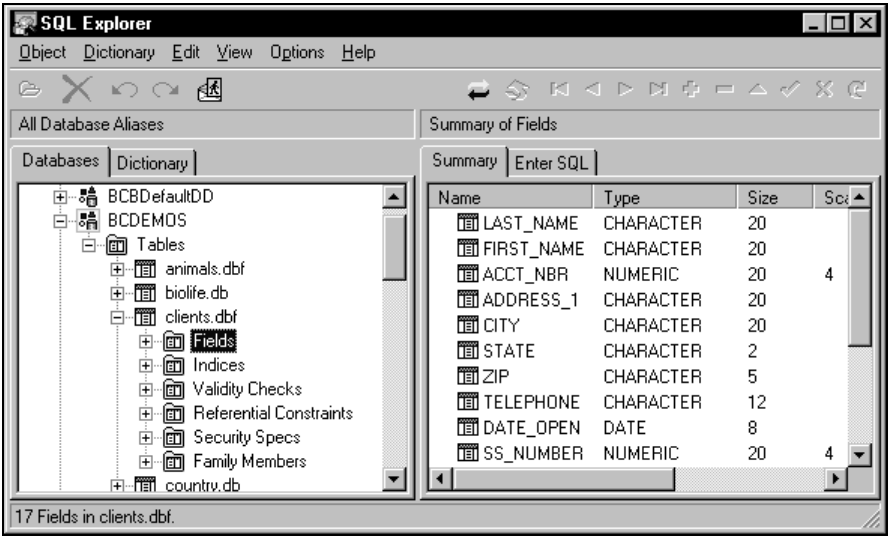


Рис. 24.31. Диалоговое окно программы SQL Explorer

SQL Explorer позволяет:

- выполнять действия с псевдонимами;
- просматривать структуру БД и таблиц;
- просматривать и редактировать записи таблиц;
- редактировать и выполнять SQL-запросы;
- выполнять операции со словарями данных.

Операции с псевдонимами не отличаются от аналогичных операций в программе BDE Administrator.

Для просмотра структуры БД (таблицы) в левой панели окна выбирается нужная БД (таблица) — на рис. 24.31 это локальная таблица clients.dbf, расположение которой указывает псевдоним `BCDEMOS`. Для этой таблицы можно просмотреть ее объекты:

- поля (**Fields**);
- индексы (**Indices**);
- ограничения на значения полей (**Validity Checks**);
- ограничения ссылочной целостности (**Referential Constraints**);
- члены семейства (**Family Members**);
- права доступа (**Security Spects**).

Список доступных объектов базы данных зависит от особенностей ее или таблицы. Например, для удаленной БД InterBase в состав ее структуры включаются триггеры и генераторы.

Если для БД или таблицы установлен режим ограничения доступа, то при попытке открыть ее (получить к ней доступ) запрашиваются имя пользователя и его пароль.

Для просмотра и редактирования содержимого таблицы нужно выделить ее название и в правой панели программы SQL Explorer выбрать вкладку **Data**. Записи таблицы отображаются в сетке, не отличающейся от компонента `DBGrid`. Для навигации по записям и их редактирования можно использовать навигационный интерфейс, расположенный в панели инструментов и аналогичный компоненту `DBNavigator`.

Для просмотра содержимого данных формата BLOB (например, рисунков) следует нажать кнопку **Explorer BLOB**, которая расположена в панели инструментов. Объект BLOB выводится в отдельном окне.

Редактирование и запуск SQL-запросов осуществляется в текстовом редакторе, расположенном на вкладке **Enter SQL**. Выполнение запроса происходит при нажатии кнопки, размещенной справа от текстового редактора. Результаты работы запроса выводятся в окне, расположенном под окном текстового редактора. Переключение между запросами выполняется при нажатии кнопок с изображениями синих треугольников: нажатие верхней кнопки вызывает переход к предыдущему запросу, а нижней — к следующему.

Для операций со словарями данных нужно выбрать вкладку **Dictionary** (Словарь) в левой части окна программы SQL Explorer (см. рис. 24.31). При этом структура выбранного словаря данных отобразится в виде иерархического дерева. Со словарем можно выполнить такие операции, как:

- выбор нового словаря (**Select**);
- включение словаря в Хранилище объектов (**Register**);
- исключение словаря из Хранилища объектов (**Unregister**);
- создание нового словаря (**New**);

- удаление выбранного словаря (**Delete**);
- вставка в словарь данных из БД (**Import from Dadabase**).

В скобках указаны команды меню вкладки **Dictionary**.

Словарь данных включает в себя описание структур БД (объект **Database**) и атрибуты (объект **Attribute Sets**).

Программа Data Pump

С помощью этой программы можно выполнить перенос данных между таблицами БД. Под *переносом* понимается не перемещение, а копирование данных из таблиц исходной БД (источника) в таблицы другой БД (приемника). В результате переноса в базе-приемнике автоматически создаются таблицы с именами копируемых таблиц источника, и в них дублируется информация этих таблиц.

Для вызова программы запускается файл datapump.exe, находящийся в одном каталоге с процессором баз данных. Программу также можно вызвать через главное меню Windows командой **Programs | Borland Delphi 7 | Datapump**.

Работа с программой напоминает работу с программой-мастером (wizard) и осуществляется по шагам. В процессе выполнения действий открываются окна в такой последовательности:

- **Select Source Alias** (Выбор псевдонима источника);
- **Select Target Alias** (Выбор псевдонима приемника);
- **Select Tables to Move** (Выбор таблиц для перемещения);
- **Inspect or Modify Items** (Проверка или изменение элементов).

Кроме того, на четвертом шаге при изменении (просмотре) структуры таблиц открываются дополнительные окна.

В каждом окне выбирается или вводится соответствующая информация. Переход к следующему или возврат к предыдущему шагам (окнам) выполняется нажатием кнопок **Next** и **Back** соответственно. Нажатие кнопки **Exit** прекращает процесс подготовки переноса данных и завершает работу программы.

Выбор источника заключается в задании псевдонима БД или в непосредственном указании диска и каталога ее размещения. Для базы-приемника также необходимо задать псевдоним. Размещение источника и приемника не могут совпадать, даже если изменить имена создаваемых таблиц так, чтобы они не совпадали с именами исходных таблиц.

Выбор таблиц заключается в указании имен главных файлов таблиц, для которых будет выполнен перенос данных. Одной операцией переноса можно выполнить копирование данных сразу для нескольких таблиц БД.

На последнем шаге выполняется управление полями выбранных таблиц. В окне, показанном на рис. 24.32, можно просмотреть или изменить структуру перемещаемых таблиц.

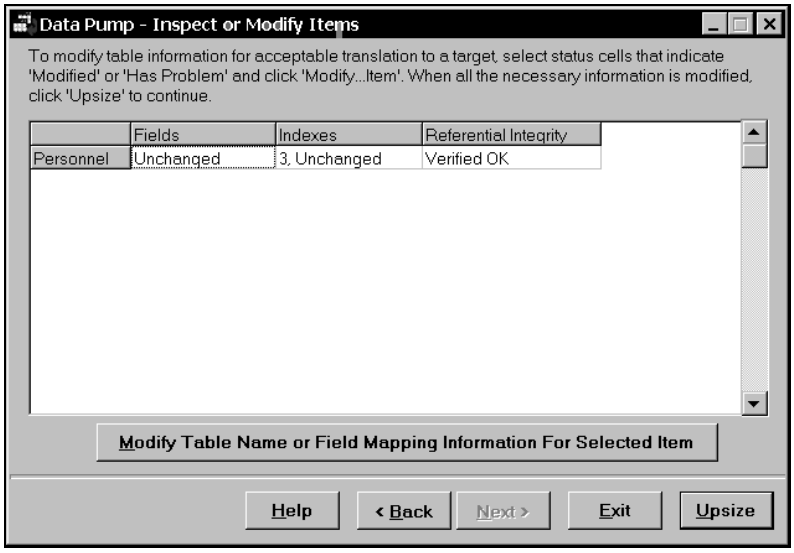


Рис. 24.32. Управление полями таблиц при переносе БД

Изменение структуры относится не к исходным таблицам, а к вновь создаваемым таблицам базы-приемника. Для новых таблиц можно изменить следующие элементы структуры:

- **Fields** — описания полей и имя таблицы;
- **Indexes** — индексы;
- **Referential Integrity** — ограничения ссылочной целостности.

Для изменения элемента таблицы его нужно выделить, тогда вокруг него отобразится пунктирный прямоугольник (на рис. 24.32 выбран элемент **Fields** таблицы `Album`, что отображается едва заметным пунктирным выделением ячейки с текстом `Unchanged`). Нажатие кнопки **Modify Table Name or Field Mapping Information For Selected Item** (Изменить имя таблицы или информацию о полях для выбранного элемента) открывает окно изменения полей **Data Pump — Modify Fields** (рис. 24.33).

Список **Source Field Names** содержит имена всех полей исходной таблицы, при этом в полях ввода **Table Name** выводятся имена исходной (столбец **Source**) и создаваемой (столбец **Target**) таблиц соответственно. По умолчанию эти имена совпадают, но имя таблицы-приемника можно изменить (поле ввода **Table Name** для таблицы-источника заблокировано).

Поля ввода (ввод заблокирован), расположенные в столбце **Source**, отображают характеристики поля таблицы, выбранного в списке полей источника. Поля ввода и списки, расположенные в столбце **Target**, содержат те же характеристики для создаваемой таблицы, там же расположено поле ввода **Field Name**, указывающее имя поля таблицы-приемника. По умолчанию описания полей таблицы (имена и характеристики) приемника совпадают с описаниями полей таблицы источника, однако при необходимости их можно изменять.

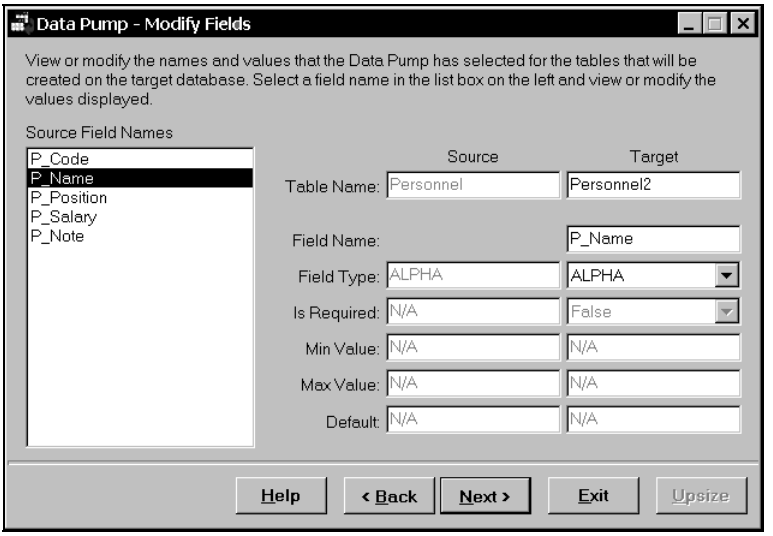


Рис. 24.33. Изменение полей и имени таблицы

Поле таблицы имеет следующие характеристики:

- **Field Type** (тип);
- **Is Required** (требование обязательного ввода значения);
- **Min Value** (минимальное значение);
- **Max Value** (максимальное значение);
- **Default** (значение по умолчанию).

Нажатие кнопки **Next** (**Back**) подтверждает (отменяет) сделанные изменения и открывает следующее (предыдущее) окно.

Изменить индексы можно только для таблиц, у которых они есть, в этом случае в столбце **Indexes** таблицы выводится число индексов. Так, на рис. 24.32 исходная таблица `Personnel` имеет 3 индекса, включая индекс, автоматически созданный по ключу. При нажатии кнопки модификации индексов вызывается окно **Data Pump — Modify an Index on Table** (рис. 24.34).

В этом окне можно просмотреть определение индексов и при необходимости изменить их, задав для индекса новые:

- имя;
- состав полей;
- порядок полей.

Напомним, что ключ таблицы Paradox не имеет имени.

Нажатие кнопки **Next** (**Back**) отображает в окне определений следующий (предыдущий) индекс таблицы. После вывода определения последнего индекса осуществляется возврат в окно **Inspect or Modify Items**.

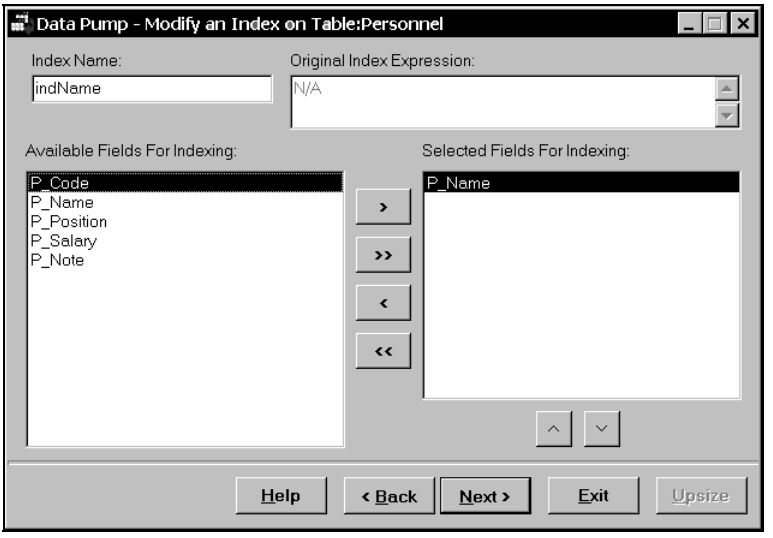
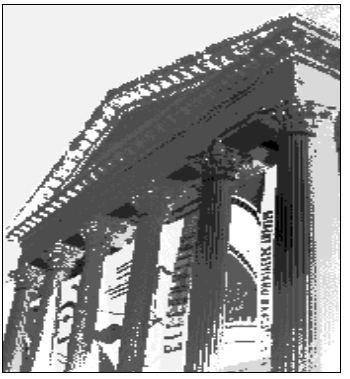


Рис. 24.34. Окно изменения индексов

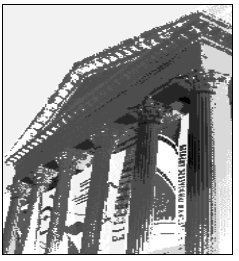
Копирование данных запускается нажатием кнопки **Upsize** (Перенос). В процессе копирования отображается индикатор, показывающий ход переноса данных, а по окончании копирования выводится отчет о его результатах. Если в базе-приемнике уже существует таблица с заданным именем, то выдается запрос на подтверждение операции.



ЧАСТЬ V

Удаленные базы данных

ГЛАВА 25



Введение в работу с удаленными базами данных

Ранее были рассмотрены локальные базы данных, когда и БД, и взаимодействующее с ней приложение располагаются на одном компьютере. В данной главе мы рассмотрим некоторые особенности работы с удаленными БД, используемыми в сети, где приложение и БД располагаются на разных компьютерах.

Основные понятия

В принципе локальную БД тоже можно использовать для коллективного доступа, т. е. в сетевом варианте. В этом случае файлы базы данных и приложение для работы с ней располагаются на сервере сети. Пользователь запускает со своего компьютера находящееся на сервере приложение, при этом у него запускается копия приложения. Можно установить приложение и непосредственно на компьютере пользователя, в этом случае приложению должно быть известно местонахождение общей БД, заданное, например, через псевдоним. Подобный сетевой вариант использования локальной БД соответствует архитектуре "файл-сервер".

Достоинствами сетевой архитектуры "файл-сервер" являются простота разработки и эксплуатации БД и приложения. Разработчик фактически создает локальную БД и приложение, которые затем просто используются в сетевом варианте. При этом не требуется дополнительное программное обеспечение для организации работы с БД.

Однако архитектуре "файл-сервер" свойственны и существенные недостатки.

- Для работы с данными используется навигационный способ доступа, при этом по сети циркулируют большие объемы данных. В результате сеть оказывается перегруженной, что является причиной ее низкого быстродействия и плохой производительности при работе с БД.
- Требуется синхронизация работы отдельных пользователей, связанная с блокировкой в таблицах тех записей, которые редактирует другой пользователь.
- Приложения не только обрабатывают данные, но и управляют самой базой данных. В связи с тем, что управление БД осуществляется с разных компьютеров, затрудняются управление доступом, соблюдение конфиденциальности и поддержание целостности БД.

Из-за этих недостатков архитектура "файл-сервер", как правило, используется в небольших сетях. Для сетей с большим количеством пользователей предпочтительным вариантом (а порой и единственным возможным) является архитектура "клиент-сервер".

Архитектура "клиент-сервер"

В сетевой архитектуре "клиент-сервер" БД размещается на компьютере-сервере сети (сервере или удаленном сервере) и называется также удаленной БД. Приложение, осуществляющее работу с этой БД, находится на компьютере пользователя. Приложение пользователя является *клиентом*, его также называют *приложением-клиентом*.

Клиент и сервер взаимодействуют следующим образом. Клиент формирует и отсылает запрос (SQL-запрос) серверу, на котором размещена БД. Сервер выполняет запрос и выдает клиенту в качестве результатов требуемые данные.

Таким образом, в архитектуре "клиент-сервер" клиент посылает запрос и получает только те данные, которые ему действительно нужны. Вся обработка запроса выполняется на удаленном сервере. К достоинствам такой архитектуры относятся следующие факторы.

- Для работы с данными используется реляционный способ доступа, что снижает нагрузку на сеть.
- Приложения не управляют напрямую базой, управлением занимается только сервер. В связи с этим можно обеспечить высокую степень защиты данных.
- В приложении отсутствует код, связанный с управлением БД, поэтому приложения упрощаются.

Отметим, что сервером называют не только компьютер, но и специальную программу, которая управляет БД. Так как в основе организации обмена данными между клиентом и сервером лежит язык SQL, такую программу еще называют *SQL-сервером*, а БД — *базой данных SQL*. В широком смысле слова под сервером понимают компьютер, программу и саму базу данных. SQL-серверами являются промышленные СУБД, такие как InterBase, Oracle, Informix, SyBase, DB2, Microsoft SQL Server и др. Каждый из серверов имеет свои преимущества и особенности, связанные, например, со структурой БД и реализацией языка SQL, которые необходимо учитывать при разработке приложения. Далее мы будем понимать под *сервером* программу (т. е. SQL-сервер), а установленную на компьютере-сервере базу данных будем называть *удаленной БД*.

При работе в архитектуре "клиент-сервер" приложение должно:

- устанавливать соединение с сервером и завершать его;
- формировать и отсылать запрос серверу, получая от него результаты выполнения запроса;
- обрабатывать полученные данные.

При этом обработка данных не имеет принципиальных отличий по сравнению с обработкой данных в локальных БД.

Сервер и удаленная БД

Удаленная БД, как и локальная, представляет собой совокупность взаимосвязанных таблиц. Однако данные этих таблиц, как правило, содержатся в одном общем файле. Как и в случае с локальной БД, для таблиц удаленной БД могут устанавливаться связи (отношения), ограничения ссылочной целостности, ограничения на значения столбцов и т. д.

Замечание

Для удаленных БД поле называется столбцом.

Для управления БД сервер использует:

- триггеры;
- генераторы;
- хранимые процедуры;
- функции, определяемые пользователем;
- механизм транзакций;
- механизм кэшированных изменений;
- механизм событий.

Многие из перечисленных элементов обеспечиваются возможностями языка SQL-сервера, в котором, по сравнению с локальной версией, имеются существенные особенности, рассматриваемые далее.

Средства работы с удаленными БД

Система Delphi обеспечивает разработку приложений для различных серверов, предоставляя для этого соответствующие средства. Отметим, что многие описанные ранее принципы разработки приложений и средства для работы с локальными БД относятся и к работе с удаленными БД. В частности, для разработки приложений используются такие компоненты, как источник данных `DataSource`, наборы данных `Table`, `ADOTable`, `SQLTable`, `IBTable`, `Query`, `ADOQuery` и `SQLQuery`, сетка `DBGrid` и др.

Замечание

Для реализации реляционного способа доступа к удаленной БД с помощью BDE необходимо использовать только средства языка SQL. Поэтому в качестве компонентов должны выбираться такие как `Query`, `StoredProc` или `UpdateSQL`. Кроме того, для набора данных нельзя использовать методы, характерные для навигационного способа доступа, например, `Next` и `Previous` для перемещения текущего указателя или `Edit`, `Insert`, `Append` или `Delete` для изменения записей.

Напомним, что если при выполнении модифицирующего БД запроса с помощью компонента `Query` не нужен результирующий набор данных, то этот запрос предпочтительнее выполнять с помощью метода `ExecSQL`. Например:

```
Query1.Close;
Query1.SQL.Clear;
```

```
Query1.SQL.Add('DELETE FROM Personnel WHERE Position = 'Менеджер');
Query1.ExecSQL;
```

Для работы с таблицами и запросами по-прежнему можно использовать такие программы, как Database Desktop и SQL Explorer.

Средства Delphi, предназначенные для работы с удаленными БД, можно разделить на два вида:

- инструменты;
- компоненты.

К *инструментам* относятся специальные программы и пакеты, обеспечивающие обслуживание БД вне разрабатываемых приложений. Среди них:

- InterBase Server Manager — программа управления запуском сервера InterBase;
- IBConsole — консоль сервера InterBase;
- SQL Monitor — программа отслеживания порядка выполнения SQL-запросов к удаленным БД.

Компоненты предназначены для создания приложений, выполняющих операции с удаленной БД. Перечислим наиболее важные из них:

- `Database` (соединение с БД);
- `Session` (текущий сеанс работы с БД);
- `StoredProc` (вызов хранимой процедуры);
- `UpdateSQL` (модификация набора данных, основанного на SQL-запросе);
- `DCOMConnection` (DCOM-соединение);
- компоненты страниц **ADO**, **dbExpress** и **InterBase** Палитры компонентов.

Отметим, что многие из названных компонентов, например, `Database`, `UpdateSQL` и `Session`, используются также при работе с локальными БД. Так, компонент `Database` позволяет реализовать механизм транзакций, как было показано в *главе 19*, посвященной навигационному способу доступа к данным с помощью механизма BDE. Однако наиболее часто эти компоненты применяются именно при работе с удаленными базами.

Часть компонентов, например, клиентский набор данных `ClientDataSet` и соединение с сервером `DCOMConnection`, предназначена для работы в трехуровневой (трехзвенной) архитектуре "клиент-сервер" ("тонкий" клиент) и используется для построения сервера приложений.

В основе операций, выполняемых с удаленными БД как с помощью инструментов, так и программно, лежит язык SQL. Например, при создании таблицы с помощью программы IBConsole (в ее окне **Interactive SQL**) необходимо набрать и выполнить SQL-запрос (инструкцию) `CREATE TABLE`. Если создание таблицы с помощью механизма BDE осуществляется из приложения пользователя, то для этой цели используется набор данных `Query`, который выполняет такой же запрос. Основное различие заключается в том, каким образом выполняется SQL-запрос к удаленной БД.

Итак, для удаленных БД разница между средствами, используемыми в приложении, и инструментами намного меньше, чем для локальных баз данных. Поэтому далее для

удаленных БД использование инструментов (на примере программы IBConsole) и создание приложений рассматриваются параллельно.

Сервер InterBase

Все серверы имеют похожие принципы организации данных и управления ими. В качестве примера рассмотрим работу с сервером InterBase 6.*x*, который является "родным" для Delphi. Совместно с Delphi поставляются две части сервера InterBase 6.*x*: серверная и клиентская. Несмотря на то что сервер InterBase поставляется совместно с Delphi, устанавливается он отдельно: после установки Delphi выдается запрос на установку сервера InterBase. Установка происходит в автоматическом режиме, основные файлы сервера копируются в подкаталог INTERBASE, находящийся в каталоге BORLAND. Отметим, что бесплатная пробная (trial) версия сервера доступна по адресу **www.borland.com/interbase**.

Серверная часть InterBase является локальной версией сервера InterBase и используется для отладки приложений, предназначенных для работы с удаленными БД, позволяя на одном компьютере проверить их в сетевом варианте. После отладки на локальном компьютере приложение можно перенести на сетевые компьютеры без изменений, для чего нужно:

- скопировать БД на сервер;
- установить для приложения новые параметры соединения с удаленной БД.

Скопировать БД можно с помощью программ типа Проводник Windows.

Клиентская часть нужна для обеспечения доступа приложения к удаленной БД.

При разработке БД и приложений с использованием локальной версии сервера InterBase нужно иметь в виду, что она имеет ряд ограничений и может не поддерживать, например, механизм событий сервера или определяемые пользователем функции. Полнофункциональная версия сервера InterBase приобретается и устанавливается отдельно от Delphi.

Как упоминалось, в основе работы с удаленной БД лежат возможности языка SQL, обеспечивающие соответствующие операции. Назначение и возможности языка SQL для удаленных БД в принципе совпадают с назначением и возможностями этого языка для локальных БД. Далее мы рассмотрим особенности языка SQL для удаленных БД.

При рассмотрении инструкций языка будем опускать несущественные операнды и элементы. При описании формата инструкций языка SQL используются следующие правила:

- символы < и > обозначают отдельные элементы формата инструкций, например имена таблиц и столбцов, и при записи инструкций SQL не указываются;
- в квадратные скобки заключаются необязательные элементы конструкций языка;
- элементы списка, из которого при программировании можно выбрать любой из этих элементов, разделяются знаком |, а сам список заключается в фигурные скобки.

Для наглядности зарезервированные слова языка SQL будем писать заглавными (прописными) буквами, а имена — строчными (маленькими). Регистр букв не влияет на интерпретацию инструкций языка.

Бизнес-правила

Как отмечалось, бизнес-правила представляют собой механизмы управления БД и предназначены для поддержания БД в целостном состоянии. Кроме того, они нужны для реализации ограничений БД, а также для выполнения ряда других действий, например, накапливания статистики работы с БД.

Бизнес-правила можно реализовывать на физическом и программном уровнях. В первом случае эти правила (например, ограничение ссылочной целостности для связанных таблиц) задаются при создании таблиц и входят в структуру БД. Для этого в синтаксис инструкции `CREATE TABLE` включаются соответствующие операнды, например, `DEFAULT` (значение по умолчанию). В дальнейшей работе нельзя нарушить или обойти ограничение, заданное на *физическом* уровне.

На программном уровне бизнес-правила можно реализовать в сервере и в приложении. Причем эти бизнес-правила не должны быть определены на физическом уровне. Для реализации бизнес-правил в сервере обычно используются *триггеры*. Достоинствами такого подхода является то, что вычислительная нагрузка по управлению БД целиком ложится на сервер, что снижает нагрузку на приложение и сеть, а также то, что действие ограничений распространяется на все приложения, осуществляющие доступ к БД. Однако одновременно снижается гибкость управления БД. Кроме того, нужно учитывать, что средства отладки триггеров и хранимых процедур сервера развиты недостаточно хорошо.

Для программирования бизнес-правил в приложении используются *компоненты* и их средства. Достоинство такого подхода заключается в легкости изменения бизнес-правил и возможности определить правила "своего" приложения. Недостатком является снижение безопасности БД, связанное с тем, что каждое приложение может устанавливать свои правила управления БД. В *главе 19*, посвященной навигационному способу доступа, мы рассмотрели программирование бизнес-правил в приложении на примере каскадного удаления записей в связанных локальных таблицах.

Организация данных

Информация всей БД сервера InterBase хранится в одном файле с расширением gdb. Размер этого файла может составлять единицы и даже десятки гигабайт. Отметим, что аналогичный размер БД имеет СУБД Microsoft SQL Server, в то время как для более мощных СУБД Oracle и SyBase размер БД достигает десятков и сотен гигабайт.

В отличие от локальной БД, структуру которой составляли таблицы (отдельные или связанные), удаленная БД имеет более сложную структуру, которая включает в свой состав следующие элементы:

- таблицы;
- индексы;
- ограничения;
- домены;

- просмотры;
- генераторы;
- триггеры;
- функции пользователя;
- хранимые процедуры;
- исключения;
- BLOB-фильтры;
- привилегии.

Элементы структуры удаленной БД также называют *метаданными*. Слово "мета" имеет смысл "над", т. е. метаданные представляют собой данные, которые описывают структуру БД.

Для InterBase максимальное число таблиц в БД равно 65 536, а максимальное число столбцов в таблице — 1000. Отметим, что таблицы InterBase имеют меньшее число допустимых типов столбцов (полей), чем таблицы локальных БД Paradox. Типы столбцов базы InterBase приведены в табл. 25.1.

Таблица 25.1. *Типы столбцов таблиц InterBase*

Тип	Описание
`SMALLINT`	Целое число. Диапазон –32 768...32 767
`INTEGER`	Целое число. Диапазон –2 147 483 648...2 147 483 647
`FLOAT`	Число с плавающей точкой. Диапазон по модулю 3.4×10^{-38}...3.4×10^{38}. Точность — 7 цифр мантиссы
`DOUBLE PRECISION`	Число с плавающей точкой. Диапазон по модулю 1.7×10^{-308}...1.7×10^{308}. Точность — 15 цифр мантиссы
`CHARACTER(N)`	Строка длиной `N` символов (не более 32 767)
`VARCHAR(N)` или `CHARACTER(N) VARYING`	Строка символов длиной до `N` символов (не более 32 767)
`DATE`	Дата. Диапазон 01.01.0100...11.12.5941
`BLOB`	Двоичные данные любого типа. Размер не ограничивается

В таблицах InterBase отсутствуют такие типы, как логический и автоинкрементный. Логический тип заменяется типом `CHAR(1)`, а вместо автоинкрементного типа для обеспечения уникальных значений используются генераторы и триггеры.

Запуск сервера

Для запуска сервера предназначена программа InterBase Server Manager (рис. 25.1), вызываемая одноименной командой главного меню Windows или через Панель управления. Функции управления сервером InterBase, например, управление подключением к серверу пользователей и просмотр информации о БД, в последних версиях Delphi реализует программа IBConsole.

Состояние сервера выводится в панели **Status**: запущенному состоянию сервера соответствует надпись **Running**, остановленному — **Stopped**.

Рис. 25.1. Диалоговое окно программы InterBase Server Manager

Сервер InterBase может запускаться автоматически или в ручном режиме, чем управляют переключатели группы **Startup Mode** (Режим запуска). Если выбран переключатель **Automatic**, то сервер будет автоматически вызываться при каждом запуске (перезапуске) Windows. Если же выбран ручной запуск (**Manual**), то сервер запускается нажатием кнопки **Start**. После запуска сервера кнопка **Start** изменяет свое название на **Stop**, и ее повторное нажатие приводит к остановке сервера.

В Windows NT сервер можно запустить как службу (service). Установленный флажок **Run the InterBase server as a service on Windows NT** указывает, что сервер InterBase запускается как служба Windows NT.

Панель **Root Directory** показывает главный каталог, в котором установлен сервер InterBase и который можно изменить, нажав кнопку **Change** и выбрав нужный каталог. Однако на практике изменять главный каталог не требуется.

Завершить работу сервера также можно, открыв щелчком мыши на значке контекстное меню и выбрав команду **Shutdown**.

При запущенном сервере с помощью кнопок **Server Properties** (Свойства сервера) и **Guardian Properties** (Свойства безопасности) открываются диалоговые окна настройки соответствующих свойств. Обычно в них уже заданы нужные значения, изменять которые нет необходимости.

При запуске сервера в качестве службы Windows NT управлять его параметрами, а также остановить сервер можно в окне **Services**.

Особенности приложения

Есть основные принципы разработки и использования приложений, выполняющих операции с удаленными БД. Эти принципы являются общими для различных систем, например, таких как InterBase, Microsoft SQL Server или Oracle. Поэтому примеры программ (приложений) для работы с БД InterBase, которые приводятся далее, годятся и

для других серверов. При разработке приложений использованы компоненты, стандартные для всех БД.

Наряду с этим в Delphi имеется ряд компонентов, предназначенных только для работы с сервером InterBase. Эти компоненты расположены на странице **InterBase** Палитры компонентов. Многие из них являются аналогами соответствующих компонентов страницы **BDE** и отличаются тем, что адаптированы специально под InterBase.

Особенность использования данных компонентов заключается в том, что для них доступ к БД осуществляется напрямую через процессор баз данных BDE. При этом нет необходимости в соответствующем драйвере SQL-Links.

Компоненты страницы **InterBase** принципиально не отличаются от соответствующих компонентов других страниц, поэтому здесь они описаны кратко.

Соединение с базой данных

Для выполнения любых операций с БД с ней необходимо установить соединение, т. е. *открыть* БД. По завершении работы соединение нужно разорвать, или завершить (*закрыть* БД). Для соединения с БД программы типа IBConsole имеют соответствующие средства, вызываемые с помощью команд меню. При создании приложения разработчик должен организовывать соединение самостоятельно, для чего Delphi предоставляет соответствующие компоненты, в первую очередь компонент `Database`.

Рассмотрим, как установить соединение с удаленной БД с помощью программы IBConsole и компонентов Delphi, поддерживающих механизм доступа BDE. В *главе 27* рассматривается работа с сервером InterBase по технологии InterBase Express.

Соединение с базой из программы IBConsole

Перед установлением соединения с БД нужно с помощью команд **Server | Register** и **Server | Login** выполнить регистрацию сервера InetrBase, а также задать имя и пароль пользователя.

Замечание

Заметим, что первоначально на сервере InetrBase для пользователей есть стандартное имя SYSDBA и пароль "masterkey", которые могут изменяться администратором.

Соединение с БД выполняется командой **Database | Connect**. После открытия БД доступна для работы, например, для изменений структуры путем добавления и удаления таблиц или для редактирования данных.

Для отключения от БД следует выполнить команду **Database | Disconnect**, при этом запрашивается подтверждение на выполнение этой операции. При утвердительном ответе соединение с БД разрывается. При необходимости также запрашивается подтверждение текущей незавершенной транзакции.

На программном уровне соединение с БД выполняет инструкция `CONNECT`, имеющая следующий формат:

```
CONNECT DATABASE "<Имя файла БД>"
USER "<Имя пользователя>" PASSWORD "<Пароль пользователя>"
```

Отключение от БД выполняет инструкция `EXIT`.

Компонент *Database*

Компонент `Database` служит для соединения с БД. Если программист при разработке приложения не разместил в форме компонент `Database`, то для организации соединения с БД в процессе выполнения приложения будет создаваться динамический (временный) объект типа `TDatabase`. Динамическое создание этого объекта выгодно тем, что не требует от программиста каких-либо действий. Однако при использовании динамического объекта типа `TDatabase` он автоматически создается для каждого соединения с БД, что при одновременной работе с несколькими БД требует бóльших ресурсов Windows, чем в случае применения компонента `Database`.

Использование компонента `Database` позволяет:

- настраивать параметры соединения с БД;
- явно управлять транзакциями при работе с БД.

Компонент `Database` можно использовать для локальных и для удаленных БД. Например, действия, связанные с настройкой псевдонимов BDE, одинаковы для обоих типов БД.

Свойство `SessionName` типа `String` указывает компонент сеанса `Session`, с которым связан компонент `Database`. Если значение этого свойства не задано (пустое значение или значение `Default`), то для сеанса создается динамический объект типа `TSession` (объект по умолчанию).

Свойство `AliasName` типа `String` указывает псевдоним БД. На этапе разработки приложения псевдоним выбирается из списка в окне Инспектора объектов.

Расположение БД может быть определено также с помощью свойства `DriverName` типа `String`, задающего драйвер БД. Следует иметь в виду, что значения этого свойства и свойства `AliasName` являются взаимоисключающими: при установке одного из них второе автоматически сбрасывается. Если расположение БД задано значением свойства `DriverName`, то остальные параметры соединения должны быть заданы через свойство `Params`.

Свойство `DatabaseName` типа `String` задает имя БД, используемое в приложении для соединения с БД. Отметим, что это имя не совпадает ни с псевдонимом, ни с собственно именем БД (именем главного ее файла). Имя БД, заданное свойством `DatabaseName`, действует только в приложении и только для организации подключения к БД, указанной ее псевдонимом.

Впоследствии набор данных (обычно `Query`) связывается с БД через свойство `DatabaseName`, значение которого должно совпадать со значением одноименного свойства компонента `Database`, используемого для соединения. На этапе разработки приложения имя БД для набора данных выбирается в окне Инспектора объектов в списке, содержащем его наряду с псевдонимами BDE.

Свойство `Params` типа `TStrings` определяет параметры соединения, для изменения которых при разработке приложения используется редактор списка значений Value List editor, вызываемый двойным щелчком в области значения Инспектора объектов. С его помощью можно (неясно, почему) установить значение одного параметра: в поле **Key** указывается имя параметра, а в поле **Value** — значение параметра.

Нажатие кнопки **Code Editor** преобразует список к текстовому виду и отображает его на отдельной вкладке в окне Редактора кода (рис. 25.2). В окне редактора кода можно задавать и редактировать произвольное число параметров. Более того, теперь при двойном щелчке в области значения свойства `Params` Инспектора объектов всегда происходит вызов Редактора кода, а не редактора Value List editor.

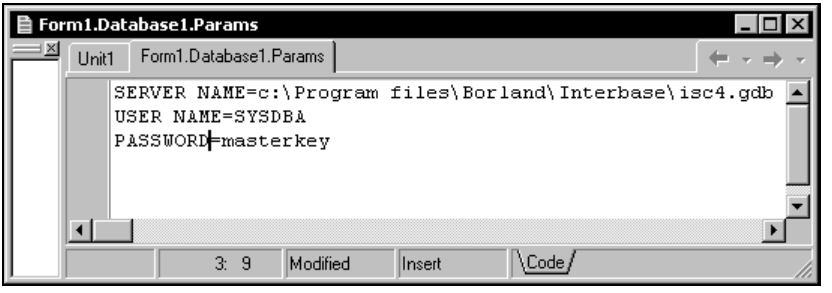


Рис. 25.2. Параметры соединения с БД в окне Редактора кода

В приведенном на рис. 25.2 примере указаны имя сервера, имя пользователя и его пароль. Отметим, что имя сервера лучше задавать через псевдоним, что облегчает перенос БД на другое место.

При выполнении приложения доступ к отдельному параметру осуществляется по его индексу в массиве (списке) параметров `Params`. При этом необходимо учитывать, что в значение параметра также входит его название. Рассмотрим следующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.text := Database1.Params[1];
end;
```

Здесь при нажатии кнопки в поле редактирования (компонент `Edit`) выводится второй параметр соединения.

В частности, для параметров, приведенных на рис. 25.2, будет получена следующая строка:

```
USER NAME = SYSDBA
```

Кроме параметров, заданных в свойстве `Params` компонента `Database`, соединение с БД также определяется:

- установками системных параметров BDE;
- параметрами драйвера, используемого для доступа к БД;
- параметрами псевдонима БД.

Если в списке Params какой-либо параметр, например LANGDRIVER, не задан, он выбирается из перечисленных выше установок. Поэтому с помощью свойства Params нужно устанавливать (переустанавливать) только те параметры, значения которых не устраивают разработчика.

Свойство LoginPrompt типа Boolean управляет режимом отображения окна ввода имени пользователя и пароля **Database Login** (рис. 25.3). По умолчанию свойство имеет значение True, и окно появляется при первом соединении с БД. Если установить свойство LoginPrompt в значение False, то запрос на идентификацию пользователя не выдается, и его имя и пароль должны быть указаны в параметрах соединения (свойство Params).

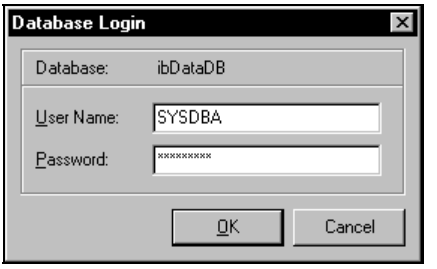


Рис. 25.3. Окно ввода имени пользователя и пароля

Свойство KeepConnection типа Boolean определяет, сохранять ли соединение с БД, если с ней не связан ни один набор данных. По умолчанию свойство имеет значение True, и однократно установленное соединение сохраняется в течение всей работы с БД. Использование постоянного соединения с БД экономит время, затрачиваемое на доступ к данным, но требует определенных системных затрат на его поддержание. Постоянное соединение разрывается при окончании работы приложения или при программной установке свойства Connected в значение False.

На этапе разработки управлять значениями названных свойств можно в окне Редактора параметров соединения (рис. 25.4). Оно вызывается двойным щелчком на компоненте Database или вызовом команды **Database editor** контекстного меню компонента.

Элементы управления этого окна отображают и позволяют изменять значения соответствующих свойств компонента Database:

- поле редактирования **Name** — свойство DatabaseName;
- комбинированный список **Alias name** — свойство AliasName;
- комбинированный список **Driver name** — свойство DriverName;
- многострочное поле редактирования **Parameter overrides** (Новые значения параметров) — свойство Params;
- флажок **Login prompt** (Входное сообщение) — свойство LoginPrompt;
- флажок **Keep inactive connection** (Сохранять неактивное соединение) — свойство KeepConnection.

Отметим, что на рис. 25.3 и 25.4 в качестве имени базы данных выводится именно значение свойства DatabaseName, а не имя (значение свойства Name) компонента Database.

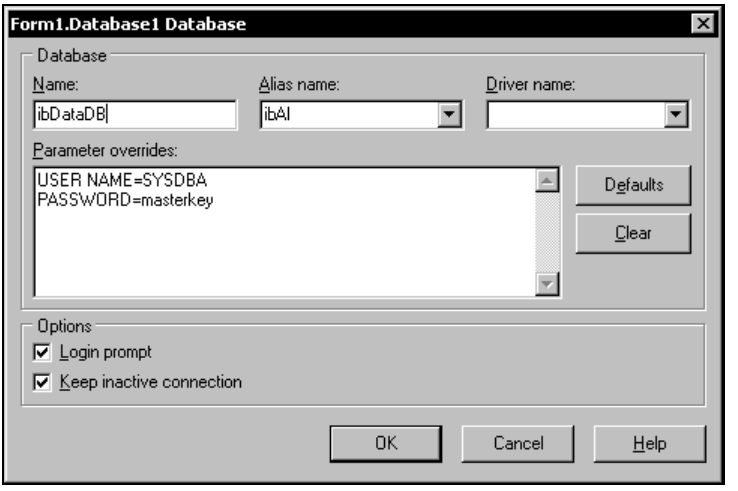


Рис. 25.4. Редактор параметров соединения с БД

Свойство `Connected` типа `Boolean` определяет, установлено ли соединение с БД. Соединение с БД можно установить при разработке приложения, задав для этого свойства значение `True` через Инспектор объектов.

При выполнении приложения можно также установить или разорвать соединение с БД, используя методы `Open` и `Close` соответственно.

Свойства `DataSetCount` и `DataSets` позволяют получить доступ к наборам данных, связанным в настоящий момент времени с БД. Свойство `DataSetCount` типа `Integer` определяет число таких наборов, а свойство `DataSets[Index: Integer]` типа `TDBDataSet` представляет собой коллекцию (массив) этих наборов данных. Доступ к отдельному набору данных можно получить по его номеру в массиве (нумерация начинается с нуля, т. е. последний набор данных имеет номер `DataSetCount – 1`).

При выполнении операций с удаленными БД автоматически запускается механизм транзакций, в данном случае называемый *неявным*. Кроме того, программист может управлять транзакциями и явно, но при этом нужно учитывать, что явные транзакции несколько замедляют выполнение операций с БД. Механизм транзакций реализуется с помощью ряда свойств и методов компонента `Database`.

Методы `StartTransaction` (инициирует начало транзакции), `Commit` (подтверждает текущую транзакцию и `Rollback` (отменяющий ее) были рассмотрены в *главе 19*, посвященной навигационному способу доступа к данным с помощью механизма BDE.

Свойство `InTransaction` типа `Boolean` позволяет определить, существует ли активная транзакция для текущего соединения с БД. Если имеется незавершенная транзакция, то свойство имеет значение `True` и значение `False` — в противном случае. Это свойство можно использовать, например, при подтверждении транзакции. Если активной транзакции нет, то вызов метода `Commit` приводит к генерации исключения. Например:

```
if Database1.InTransaction then Database1.Commit;
```

Если имеется активная транзакция, то она подтверждается, а все сделанные в ее рамках изменения БД вступают в действие.

Свойство TransIsolation типа TTransIsolation управляет *уровнем изоляции* транзакций. Уровень изоляции определяет, каким образом транзакция взаимодействует с другими транзакциями. Далее приведен список возможных значений свойства TransIsolation.

- tiDirtyRead — разрешается чтение изменений в записях БД, сделанных в рамках других транзакций (другими пользователями). В связи с тем, что на момент чтения эти изменения являются неподтвержденными и впоследствии могут быть отменены, этот уровень изоляции обеспечивает минимальную изоляцию (защиту) от других транзакций.
- tiReadCommitted — разрешается чтение только подтвержденных изменений в записях БД (по умолчанию). Если изменения еще не подтверждены, то читается предыдущая версия записи.
- tiRepeatableRead — считывание сразу всей БД, после чего изменения в БД, сделанные в рамках других транзакций, становятся невидимыми. Этот уровень обеспечивает максимальную изоляцию.

Для локальных БД dBase и Paradox допустимым является только значение tiDirtyRead. Для сервера InterBase значение tiDirtyRead интерпретируется как tiReadCommitted.

Компонент *Session*

Компонент Session представляет собой текущий сеанс работы с БД и наряду с компонентом Database используется для управления соединением с БД. Однако если компонент Database позволяет задавать параметры одного соединения с БД или параметры соединения с одной БД, то компонент Session управляет всеми соединениями со всеми БД. Если программист при разработке приложения не разместил в форме компонент Session, то будет создаваться динамический (временный) объект типа TSession (TSessionList).

Наряду с управлением соединениями с БД, использование компонента Session также позволяет:

- управлять установками BDE;
- управлять паролями БД.

Свойство SessionName типа String задает имя сеанса. Это имя устанавливается программистом, и его назначение не совпадает с назначением свойства Name, которое определяет имя самого компонента Session. Однако в частном случае эти имена могут совпадать.

Свойства сеанса, заданного компонентом Session, действуют на все связанные с ним компоненты Database. Напомним, что компонент Database связывается с сеансом через свое свойство SessionName, значение которого при разработке приложения выбирается в списке Инспектора объектов.

С помощью свойств DatabaseCount и Databases можно получить доступ к активным БД, с которыми в настоящее время установлено соединение. Свойство DatabaseCount типа Integer содержит число таких БД, а свойство Databases[Index: Integer] типа TDatabase представляет собой коллекцию (массив) объектов Database, используемых

для соединения с этими БД. Доступ к отдельной БД можно получить по ее номеру в массиве (нумерация начинается с нуля, поэтому последняя БД имеет номер `DatabaseCount – 1`).

Свойство `KeepConnections` типа `Boolean` определяет, сохранять ли соединение с БД, если с ней не связан ни один набор данных. Это свойство аналогично свойству `KeepConnection` соединения с отдельной БД, заданного компонентом `Database`. Разорвать соединение с БД можно, вызвав метод `DropConnections`.

Свойства `NetFileDir` и `PrivateDir` типа `String` позволяют задать каталоги для хранения управляющего сетевого файла с расширением net (для таблиц Paradox) и рабочего каталога для хранения временных файлов сеанса соответственно.

Свойство `Active` типа `Boolean` управляет активностью сеанса. Установка этого свойства в значение `True` активизирует сеанс, а значения `False` — деактивизирует его. При выполнении приложения управлять активностью сеанса можно также с помощью методов `Open` и `Close` соответственно.

Непосредственно перед активизацией сеанса генерируется событие `OnStartup` типа `TNotifyEvent`.

Для управления установками BDE компонент `Session` имеет средства, позволяющие читать и изменять параметры драйверов и псевдонимов БД. Рассмотрим наиболее важные из них.

Методы `GetAliasNames` и `GetAliasParams` обеспечивают получение информации о псевдонимах BDE. Процедура `GetAliasNames(List: TStrings)` позволяет получить список псевдонимов BDE. Список заносится в объект, заданный параметром `List` типа `TStrings`, которым может быть, например, список строк компонента `ListBox`. Приведем соответствующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Session1.GetAliasNames(ListBox1.Items);
end;
```

Здесь нажатие кнопки `Button1` приводит к загрузке в компонент `ListBox1` списка псевдонимов BDE. Предыдущее содержимое компонента `ListBox1` теряется.

Процедура `GetAliasParams(const AliasName: String; List: TStrings)` возвращает параметры конкретного псевдонима, имя которого задано параметром процедуры `AliasName`. Если имя псевдонима указано неправильно, то генерируется исключение. Например:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Session1.GetAliasParams(Edit1.Text, ListBox1.Items);
end;
```

При нажатии кнопки `Button2` в компонент `ListBox1` заносятся параметры псевдонима, введенного в компонент `Edit1`. Предыдущее содержимое компонента `ListBox1` теряется.

Для добавления и удаления псевдонимов предназначены методы `AddAlias` и `DeleteAlias` соответственно. Процедура `AddAlias(const Name, Driver: String; List:`

TStrings) добавляет новый псевдоним с именем Name. Параметр Driver определяет драйвер нового псевдонима, а параметр List содержит список параметров псевдонима. Пример создания псевдонима для локальной БД:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  Session1.AddAlias(Edit1.Text, 'STANDARD', Memo1.Lines);
end;
```

В процедуре создается новый псевдоним, имя которого набрано в компоненте Edit1. Псевдоним предназначен для работы с локальными БД dBase или Paradox, на что указывает тип STANDARD псевдонима. Напомним, что локальные БД этих типов работают под управлением драйвера STANDARD. Параметры псевдонима должны быть указаны в компоненте Memo1. Необходимо указать хотя бы расположение БД, например, введя в поле редактирования следующий текст:

```
PATH = d:\data
```

Имя нового псевдонима должно быть уникальным: если указанный псевдоним уже существует, то генерируется исключение.

В качестве примера рассмотрим следующую процедуру:

```
procedure TForm1.Button3Click(Sender: TObject);
var Params: TStringList;
begin
  Params := TStringList.Create;
  try
    Params.Add('SERVER NAME=d:\ibData\registration.gdb');
    Params.Add('USER NAME=SYSDBA');
    Params.Add('PASSWORD=masterkey');
    Session1.AddAlias('ibData', 'INTRBASE', Params);
  finally
    Params.Free;
  end;
end;
```

В ней создается псевдоним ibData для работы с удаленной БД InterBase. Для БД заданы три параметра: путь, имя и пароль пользователя. Для определения параметров в приложении динамически создается список Params типа TStringList. Заполнение и удаление списка также выполнено динамически. При заполнении списка и создании псевдонима выполняется обработка исключений. При их возникновении в любом случае будет удален динамически созданный список Params.

Процедура DeleteAlias(const Name: String) удаляет псевдоним, указанный параметром Name. Например:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  Session1.DeleteAlias(Edit1.Text);
end;
```

Здесь удаляется псевдоним, имя которого указано в компоненте Edit1. Если указанный псевдоним отсутствует, то процедура DeleteAlias не выполняет никаких действий.

Методы GetDriverNames и GetDriverParams обеспечивают получение информации о драйверах BDE и их параметрах соответственно, а методы AddDriver и DeleteDriver — добавление и удаление драйверов.

Соединение с базой данных из приложения

Для соединения с БД из приложения разработчик должен создать соответствующую форму. В целом создание формы приложения для работы с удаленной БД не отличается от процесса создания формы для работы с локальной БД. Визуальные и невизуальные компоненты размещаются в форме и связываются между собой обычным способом.

Пусть в форме размещены сетка DBGrid1, источник данных DataSource1 и набор данных Query1. (Напомним, что в качестве набора данных следует выбирать компонент Query и ему подобные, а не компонент Table.)

Отличие формы для работы с удаленной БД заключается в том, что дополнительно организуется соединение с БД. С этой целью необходимо:

1. Создать псевдоним БД.
2. Разместить в форме компонент Database, установить нужные значения его свойств и связать с набором данных.

Псевдоним для работы с удаленной БД удобно создать с помощью рассмотренной ранее программы BDE Administrator. Для сервера InterBase псевдоним имеет тип INTRBASE. Для псевдонима нужно задать следующие параметры:

- имя пользователя USER NAME (SYSDBA);
- расположение БД SERVER NAME (для локального сервера InterBase это путь к БД, для удаленного сервера этот путь прописан в файле hosts, размещенном в каталоге Windows);
- языковой драйвер LANGDRIVER (Pdox ANSI Cyrillic).

Для примера создадим псевдоним ibData для доступа к БД registration.gdb, размещенной в каталоге D:\ibData. Параметры псевдонима показаны на рис. 25.5.

Собственно соединение с БД выполняется с помощью компонента Database, рассмотренного в предыдущем разделе. Разместим в форме компонент Database1. Для соединения с БД, обозначенной псевдонимом ibData, свойства компонентов Database1 и Query1 нужно установить в значения, показанные в табл. 25.2.

Таблица 25.2. *Свойства компонентов, используемых для соединения с БД*

Компонент	Свойство	Значение	Примечание
Database1	AliasName	ibData	Выбирается в списке
	DatabaseName	ibDataBD	Вводится разработчиком
Query1	DatabaseName	ibDataBD	Выбирается в списке

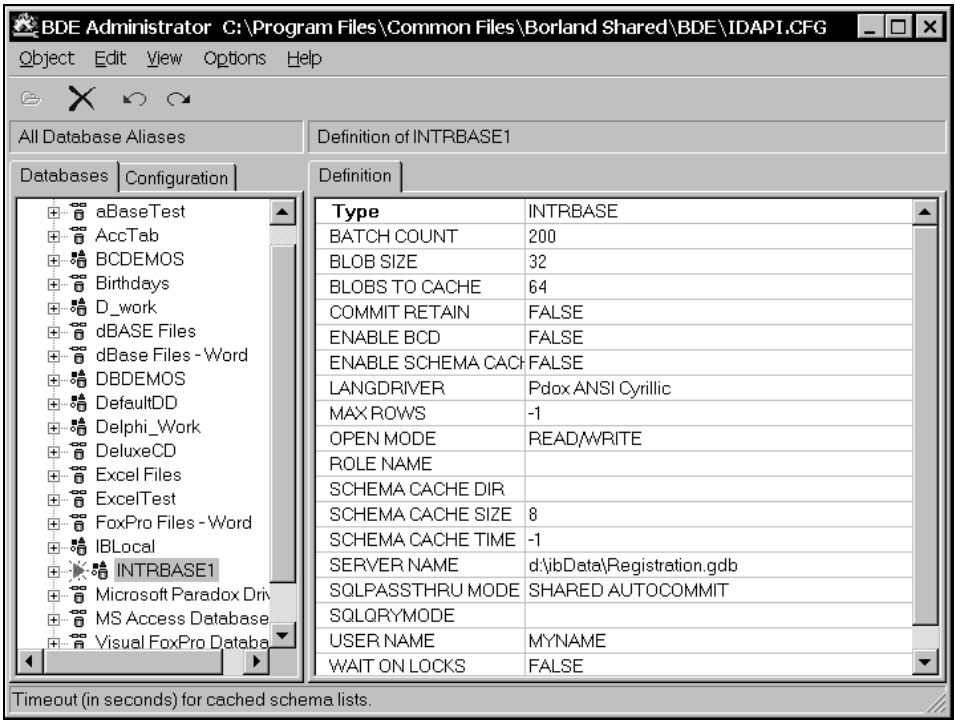


Рис. 25.5. Параметры псевдонима БД

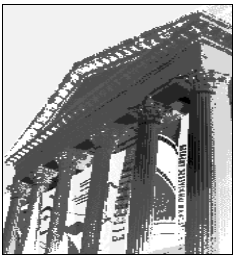
Соединение с БД устанавливается следующими способами:

- через открытие набора данных `Query1`;
- путем установки свойства `Connected` компонента `Database1` в значение `True`.

Оба способа по результатам являются эквивалентными и взаимозависимыми. Так, при открытии набора данных, например, установкой в значение `True` его свойства `Active`, свойство `Connected` компонента `Database` автоматически принимает это же значение.

Соединение с БД можно устанавливать как при разработке, так и при выполнении приложения.

ГЛАВА 26



Работа с удаленными базами данных

В данной главе мы рассмотрим основные операции, осуществляемые при работе с удаленной БД: создание БД и управление ее структурой, использование событий сервера и механизма транзакций, управление привилегиями и манипулирование данными.

Создание базы данных

В отличие от локальной БД, являющейся скорее логическим понятием, поскольку ее таблицы находятся в разных файлах и, возможно, в разных каталогах, удаленная БД представляет собой физический объект.

Для создания удаленной БД InterBase удобно использовать программу IBConsole, на примере работы с которой мы и изучим операции с БД. Процесс создания БД начинается с вызова команды **Database | Create Database**, открывающей окно **Create Database** (рис. 26.1).

Надпись **Server** отображает название сервера InterBase, в нашем случае — это локальный сервер (**Local Server**).

Для новой БД необходимо указать ее псевдоним, файлы и параметры. Псевдоним, задаваемый в поле редактирования **Alias**, предназначен для идентификации БД при работе внутри сервера InterBase и не связан с псевдонимами BDE. В списке **File(s)** перечисляются файлы создаваемой БД и их размеры (в страницах), достаточно указать один файл. Для файла необходимо задать его точное расположение, для локального сервера — это каталог.

Среди параметров наибольший интерес представляет установка набора, используемого по умолчанию для кодировки символов. Кодировка важна при выполнении таких операций, как сортировка строк. В Windows для символов, включающих русские буквы, применяется вариант Windows 1251 кода ANSI (**WIN1251**).

После нажатия кнопки **OK** создается БД с указанными параметрами. Новая база первоначально является пустой и не содержит ни таблиц, ни данных. Ее файл, тем не менее, имеет размер около 600 Кбайт, т. к. включает в себя разнообразную служебную информацию.

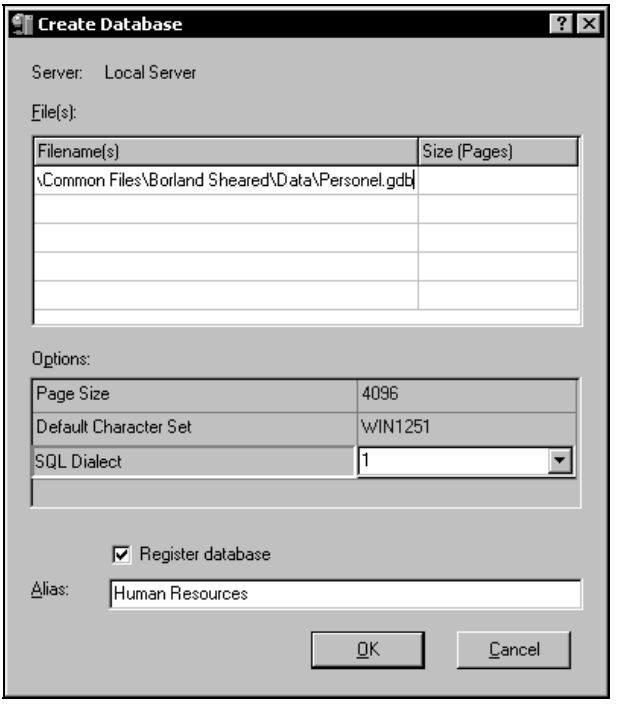


Рис. 26.1. Окно создания удаленной БД

С вновь созданной БД автоматически устанавливается соединение.

Для удаления базы данных следует выполнить команду **Database | Drop Database**, при этом появляется окно с предупреждением. После подтверждения операции происходит удаление, которое заключается в удалении с диска gdb-файлов, содержащих всю информацию базы данных.

Замечание

Восстановить удаленную БД невозможно.

Удалить БД имеет право только ее создатель или системный администратор, имеющий имя SYSDBA.

Создать и удалить БД можно также, подготовив и выполнив соответствующие запросы. Запрос можно выполнить как в среде программы IBConsole, так и из приложения с помощью компонента `Query`.

Инструкция SQL создания БД имеет следующий формат:

```
CREATE DATABASE "<Имя файла БД>"
       [USER "Имя пользователя" [PASSWORD "Пароль пользователя"]]
       [PAGE_SIZE [=] <Целое число>]
       [LENGTH [=] <Целое число> [PAGE[S]]]
       [DEFAULT CHARACTER SET <Набор символов>]
[<Вторичный файл>];
```

```
<Вторичный файл> = FILE "<Имя файла БД>" [<Файловая информация>]
       [<Вторичный файл>]

<Файловая информация> = LENGTH [=]<Целое число> [PAGE[S]] |
      STARTING [AT [PAGE]] <Целое число> [<Файловая информация>]
```

Имя файла БД указывает спецификацию (имя и путь) физического файла, в котором будет храниться информация создаваемой БД. Имя файла является единственным обязательным параметром, который должен быть задан для новой БД. Остальные параметры являются факультативными.

Если данные хранятся более чем в одном физическом файле, то говорят о *многофайловой БД*. Такая структура может быть использована при большом объеме данных, когда файл базы (с расширением gdb) достигает значительных размеров. При этом первый файл БД называется *первичным*, а все последующие — *вторичными*. Файлы могут отличаться именами, расширениями и расположением. Вторичные файлы можно размещать в других каталогах и на других дисках сервера. Размеры (длины) файлов измеряются в страницах.

Для файлов БД задаются следующие параметры:

- `PAGE_SIZE` — размер страницы в байтах, допустимые значения: 1024 (по умолчанию), 2048, 4096 и 8192; параметр задается для первичного файла и действует на все файлы БД;
- `LENGTH` — длина файла в страницах (по умолчанию 75);
- `STARTING` — страница, начиная с которой располагается файл (этот параметр фактически является альтернативой параметру `LENGTH`; если для предшествующего вторичного файла не указана длина, то должна быть указана начальная страница).

При задании размера страницы следует учитывать, что за одно обращение к БД считывается одна страница. Если при работе с БД считывается большой объем информации (много записей), то малый размер страницы увеличит число операций чтения. Кроме того, может увеличиться глубина индекса, что также нежелательно. В то же время большой размер страницы при малом объеме информации приводит к чтению лишних данных. В обоих случаях увеличивается время доступа к данным.

Например, инструкция

```
CREATE DATABASE "D:\IBDATA\personnel.gdb";
```

создает однофайловую БД с именем personnel.gdb, расположенную в каталоге D:\IBDATA.

В свою очередь, инструкция

```
CREATE DATABASE "D:\STORE\Store.gdb"
      FILE " D:\STORE\Store.gd2" STARTING AT PAGE 501 LENGTH = 300
      FILE " D:\STORE\Store.gd3";
```

приводит к созданию многофайловой БД, при этом ее данные размещаются в трех файлах с одинаковым именем Store. Файлы располагаются в общем каталоге D:\STORE и отличаются расширениями. Длина первичного файла (gdb) составляет 500 страниц и задается косвенным путем — через указание номера начальной страницы следующего

(вторичного) gd2-файла, равного 501. Длина первого вторичного файла задана равной 300 страницам, а длина второго не ограничивается.

Отметим, что при задании параметров файлов некоторые элементы являются необязательными — можно опускать знак равенства (при указании длины) и слова `AT PAGE` (при указании начальной страницы).

Параметры `USER` и `PASSWORD` определяют имя и пароль пользователя, создавшего БД. Эти параметры важны при последующем удалении БД, поскольку, как отмечалось, удалить БД может только системный администратор или создатель этой базы данных. Например, с помощью инструкции

```
CREATE DATABASE "D:\IBDATA\test.gdb" USER "User1" PASSWORD "Secret";
```

пользователь с именем `User1` и паролем `Secret` создает однофайловую базу данных с названием `test.gdb`.

Операнд `DEFAULT CHARACTER SET` позволяет определить набор, используемый для кодировки символов. Как отмечалось, для нашей страны рекомендуется задавать набор `WIN1251`. Например:

```
CREATE DATABASE "D:\IBDATA\test.gdb" DEFAULT CHARACTER SET WIN1251;
```

Здесь для БД `test.gdb` по умолчанию устанавливается набор символов `WIN1251`. Отметим, что набор символов можно также отдельно определить для каждого столбца таблицы с помощью аналогичного параметра. Однако удобнее задать набор символов сразу для всей БД, в этом случае для ее таблиц по умолчанию будет использована указанная кодировка.

Инструкция удаления БД имеет следующий вид:

```
DROP DATABASE;
```

После ее выполнения удаляется текущая БД, с которой установлено соединение.

Управление структурой таблиц

Структуру таблицы определяют следующие элементы:

- описания столбцов;
- ограничения столбцов;
- описания ключей;
- описания индексов;
- ограничения таблицы.

Описания и ограничения столбцов объединяют также под общим термином "описание столбцов". В InterBase ключи могут быть первичными и уникальными. Ограничения таблицы включают в себя ограничения ссылочной целостности и ограничения на значения столбцов.

Отметим, что ограничения таблицы, описания ключей и индексов относятся также к структуре базы данных, поэтому их можно просмотреть при выводе метаданных БД, например, с помощью программы IBConsole.

При управлении структурой таблицы удаленной БД состав и использование инструкций определения данных в языке SQL в принципе не отличаются от ранее рассмотрен-

ной версии этого языка для локальных БД. Однако в дополнение к локальной версии для столбцов таблицы удаленной БД можно задавать различные ограничения, например, значение по умолчанию или диапазон возможных значений.

Создание таблицы заключается в задании имени и структуры таблицы и выполняется с помощью инструкции CREATE TABLE:

```
CREATE TABLE <Имя таблицы> [EXTERNAL [FILE] "<Имя файла>"]
      (<Имя столбца1> <Описание столбца1> [<Ограничения столбца1>,]
      ...
       [<Имя столбцаN> <Описание столбцаN> [<Ограничения столбцаN>],]
       [<Ограничение1 таблицы>,]
      ...
       [<ОграничениеN таблицы>,]
       [<Описание ключа1>,]
      ...
       [<Описание ключаN>,]
       [<Описание индекса1>,]
      ...
       [<Описание индексаN>]);
```

Обязательно должны быть заданы имя таблицы и как минимум один столбец. Часть элементов структуры БД — ограничения столбца и описание ключей — можно специфицировать либо на уровне столбца, либо на уровне таблицы. В первом случае текст описания указывается в описании соответствующего столбца, во втором случае — отдельно, после описания всех столбцов.

По умолчанию таблица добавляется к текущей БД. Если таблицу нужно разместить не в файле БД, а в другом файле, то его имя указывается в операнде EXTERNAL.

Удаление таблицы выполняется так же, как в случае локальной БД — инструкцией DROP TABLE, имеющей формат:

```
DROP TABLE <Имя таблицы>;
```

Замечание

Удалить таблицу, для которой есть подчиненные таблицы, невозможно. Для удаления главной таблицы нужно предварительно удалить ограничения ссылочной целостности, связывающие эти таблицы.

Структуру таблицы можно изменять путем удаления или добавления столбцов, ключей, индексов и ограничений.

Замечание

Для изменения структуры таблиц нужно иметь соответствующие права (полномочия) доступа к БД и ее таблицам. Кроме того, изменяемые элементы структуры не должны в этот момент использоваться другими пользователями. Например, нельзя удалить индекс, который в данный момент применяется для поиска записей.

Как и для локальных БД, изменение таблицы выполняется инструкцией ALTER TABLE, которая позволяет добавлять и удалять отдельные столбцы, а также ограничения. В отличие от инструкции ALTER TABLE, другие инструкции типа ALTER xxx (например,

ALTER TRIGGER) удаляют предыдущее описание указанного объекта и заменяют его на новое.

Удаляя и добавляя столбцы, можно требуемым образом изменить состав таблицы. Однако следует иметь в виду, что после удаления столбца теряется вся его информация. Поэтому в случае, когда необходимо только немного изменить описание столбца, рекомендуется поступать так:

1. Создать новый столбец с требуемым описанием.
2. Скопировать данные из старого столбца, описание которого изменяется, в новый столбец.
3. Удалить старый столбец.

Рассмотрим пример, иллюстрирующий приведенную последовательность действий:

```
CREATE TABLE List
      (Name VARCHAR(15));
      ...
ALTER TABLE List
      ADD Name2 VARCHAR(20);
UPDATE List
      SET Name2 = Name;
ALTER TABLE List
      DROP Name;
```

Для ведения списка создана таблица List, имеющая столбец Name строкового типа длиной не более 15 символов. В процессе работы с таблицей возникла необходимость увеличить длину столбца до 20 символов, что и было сделано выполнением последовательности трех инструкций. Столбец получил новое имя Name2.

Замечание

При уменьшении длины столбца возможна потеря части информации из-за усечения значений, расположенных в конце более длинного столбца.

В рассмотренном способе изменения описания столбца имя нового столбца отличается от старого, что может оказаться нежелательным. В этом случае следует повторить указанные действия еще раз, присвоив новому столбцу прежнее имя.

Описание столбца

Описание столбца имеет формат:

```
<Описание столбца> =
      {<Имя столбца> | COMPUTED [BY] (<Выражение>) | <Домен>}
```

Столбец можно определить следующими тремя способами.

- Задать тип столбца, например, DATE или INTEGER, при этом создается обычный столбец указанного типа. Типы столбцов InterBase описаны далее в этой главе.
- Создать вычисляемый столбец, задав в операнде COMPUTED выражение.
- Создать столбец на основе домена.

Вычисляемый столбец содержит значения, которые рассчитываются на основе заданного выражения. Такой столбец удобно использовать, когда его значения зависят от каких-либо других значений, в том числе от значений других столбцов записи, и вид зависимости заранее известен. Тип вычисляемого столбца автоматически определяется типом результата выражения. Например, если результат выражения имеет целочисленный тип, то вычисляемый столбец также будет целочисленным.

Домен представляет собой предварительное описание столбца, которое затем можно использовать для описания столбцов. Создание домена также рассматривается в данной главе немного позже.

Пример:

```
CREATE TABLE List
      (Name      VARCHAR(20),
      Price      FLOAT,
      Number     INTEGER,
      PriceAll   COMPUTED BY (Price * Number));
```

Здесь создается таблица `List`, в которой ведется учет продажи товаров. Столбец `Name`, предназначенный для названия товара, имеет строковый тип (длина не более 20 символов). Столбец `Price` содержит цену единицы товара и имеет вещественный тип, а целочисленный столбец `Number` указывает количество товара. Столбец `PriceAll` содержит общую стоимость всего товара, которая рассчитывается как произведение цены товара на его количество. Исходя из типа операндов, тип этого вычисляемого столбца автоматически будет определен как `FLOAT`.

Ограничения столбца

Ограничения являются необязательными элементами, однако их использование позволяет автоматизировать процесс ввода значений, предотвращать ошибки ввода и управлять порядком сортировки строк.

Ограничения столбца имеют следующий формат:

```
[DEFAULT {<Значение> | NULL | USER}]
[NOT NULL]
[COLLATE <Порядок сортировки>]
[CHECK <Условия>]
```

Операнд `DEFAULT` определяет для столбца значение по умолчанию, которое автоматически заносится в столбец при добавлении к таблице новой записи.

В качестве значения по умолчанию можно указать:

- константу (литерал) — в столбец заносится указанное значение (должно иметь тип, совместимый с типом столбца);
- `NULL` — в столбец заносится нулевое значение;
- `USER` — в столбец заносится имя текущего пользователя (для столбцов строковых типов).

Операнд `NOT NULL` указывает, что столбец не может быть пустым и в обязательном порядке должен содержать значение допустимого типа и диапазона.

Замечание

При программировании инструкций нельзя допускать взаимоисключающие конструкции, например, одновременно указывать операнды DEFAULT NULL и NOT NULL.

Операнд COLLATE определяет порядок сортировки строковых значений.

Операнд CHECK позволяет установить для столбца разнообразные условия, управляющие его значениями. При редактировании записей столбцу разрешается присваивать только те значения, которые удовлетворяют заданным условиям. Эти условия могут быть как простым сравнением, так и сложной комбинацией нескольких критериев.

Операнд CHECK имеет следующий формат:

```
CHECK
   {<Выражение> <Операция сравнения > {<Выражение> | (<Отбор1>)}
   | <Выражение> [NOT] <Мин. значение> AND <Макс. значение>
   | <Выражение> [NOT] LIKE <Выражение1> [ESCAPE <Выражение2>]
   | <Выражение> [NOT] IN
    (<Выражение1> [, <Выражение2>] ... [, <ВыражениеN>] | (<ОтборM>))
   | <Выражение> IS [NOT] NULL
   | <Выражение> [NOT] {<Операция сравнения>|ALL|SOME|ANY} (<ОтборM>)
   | EXISTS (<ОтборM>)
   | SINGULAR (<ОтборM>)
   | <Выражение> [NOT] CONTAINING <Выражение1>
   | <Выражение> [NOT] STARTING [WITH] <Выражение1>}
```

Условия, ограничивающие возможные значения столбца, по сути и синтаксису идентичны условиям, задаваемым при отборе записей в инструкции SELECT, и здесь подробно не рассматриваются. Более того, при задании условий в операнде CHECK можно использовать инструкцию SELECT. Как и в инструкции SELECT, условия можно объединять (комбинировать) с помощью логических операций NOT, OR и AND.

Выражение должно иметь тип, совместимый с типом столбца, например, числовой или строковый. В выражении используются операнды, знаки операций (операторы), круглые скобки и функции. Операндами могут быть значения и имена столбцов. Состав функций (SUM, MAX, MIN, UPPER и др.) практически не отличается от состава функций, включенных в локальную версию языка SQL. Отметим функцию GEN_ID, которая возвращает уникальное значение генератора и служит для установки значений ключа. В частном случае выражение представляет собой имя столбца или значение.

Элементы, обозначенные как Отбор, представляют собой совокупность значений, отобранных с помощью инструкции SELECT. Условие отбора записей, заданное в этой инструкции, для элемента Отбор1 должно возвращать одно значение или ни одного, а для элемента ОтборM — несколько значений, в частном случае ни одного. Инструкция SELECT составляется по обычным правилам и служит для отбора записей из различных таблиц.

Чтобы задать диапазон ограничений для значений столбца, используется операция сравнения или конструкция вида

```
BETWEEN <Минимальное значение> AND <Максимальное значение>
```

Например:

```
CREATE TABLE Test
      (Name       VARCHAR(20) NOT NULL,
      Price       FLOAT CHECK (Price > 0),
      Number      INTEGER CHECK (Number BETWEEN 1 AND 1000));
```

Для таблицы `Test` значение столбца `Name` не может быть пустым, значение столбца `Price` должно быть положительным, а значение столбца `Number` — находиться в диапазоне `1..1000`.

С помощью условия можно обеспечить для столбца установку одного из значений, содержащихся в определенном наборе (списке). Набор значений бывает фиксированным (статическим) и задается непосредственно в конструкции `IN` или формируется динамически с помощью инструкции `SELECT`. В последнем случае состав набора зависит от результата работы инструкции `SELECT`, которая выполняется каждый раз при модификации записей таблицы.

Например, инструкция

```
CREATE TABLE Test2
  (Name     VARCHAR(20),
   Position VARCHAR(20) CHECK (Post IN ("Директор", "Менеджер", "Бухгалтер"),
   Attrib   VARCHAR(20) CHECK (Attrib IN (SELECT Attrib From Positions)));
```

для полей таблицы `Test2` устанавливает список возможных значений: значение поля `Position` должно входить в фиксированный набор из трех строковых констант. Недостатком подобного ограничения является то, что оно действует на уровне таблицы и в приложении не может быть изменено или отменено. Поэтому фиксированный набор обычно задают для столбцов, допустимые значения которых известны заранее и изменяются редко. В остальных случаях удобнее определять возможные значения динамически, как это сделано для поля `Attrib`: набор допустимых величин для этого поля образуется совокупностью всех значений одноименного поля таблицы `Positions`.

При необходимости можно сформировать набор значений на основании записей, удовлетворяющих определенным условиям. Например, если требуются атрибуты должностей, которым соответствует оклад, превышающий 2500 рублей, то описание поля `Attrib` будет иметь следующий вид:

```
Attrib VARCHAR(20)
  CHECK (Attrib IN (SELECT Attrib From Positions WHERE Salary > 2500)));
```

Ограничение, заданное операндом `CHECK`, указывается для конкретного столбца и располагается в строке его описания до начала описания следующего столбца. (Это остается верным и в случае, когда описание столбца занимает несколько строк.) Такое же ограничение можно установить и на уровне таблицы, разместив его после описания всех столбцов. Единственным отличием является то, что для ограничения, задаваемого на уровне столбца, можно указать имя. Операнд `CHECK`, используемый на уровне таблицы, имеет следующий формат:

```
[CONSTRAINT <Имя ограничения >]
CHECK <Условия>
```

Ограничения на значения столбцов, как и другие ограничения, именуются. Имя может быть задано при определении ограничения в операнде CONSTRAINT, но это не является обязательным: если для ограничения столбца имя не задано, то это ограничение получает имя по умолчанию.

В качестве примера рассмотрим создание таблицы с ограничениями:

```
CREATE TABLE Test5
     (Name        VARCHAR(20),
      OrderDate   DATE,
      PerformDate DATE,
      CHECK (PerformDate >= OrderDate));
```

В таблице Test5 хранится информация о заказе товара. Столбец OrderDate содержит дату заказа, а столбец PerformDate — дату выполнения заказа. Так как заказ не может быть выполнен раньше, чем он сделан, для столбца PerformDate на уровне таблицы установлено соответствующее ограничение. Имя для ограничения не задано. Это ограничение можно указать и при описании столбца PerformDate.

Добавить ограничение к уже существующей таблице можно с помощью инструкции вида

```
ALTER TABLE <Имя таблицы>
      ADD CHECK ([CONSTRAINT <Имя ограничения >] <Условия>);
```

Для удаления ограничения предназначается инструкция

```
ALTER TABLE <Имя таблицы> DROP <Имя ограничения>;
```

Можно произвести удаление и с помощью программ типа SQL Explorer.

Описание ключей

В таблицах InterBase ключи разделяются на первичные и уникальные (для главных таблиц) и внешние (для подчиненных таблиц). Первичный ключ по построению и использованию не отличается от ключа (главного ключа) рассмотренных ранее таблиц Paradox. Напомним, что ключ предназначен для обеспечения уникальности записей и ссылочной целостности БД.

Для описания первичного ключа используется операнд PRIMARY KEY формата:

```
PRIMARY KEY (<Список столбцов ключа>)
```

Имена столбцов, образующие ключ, перечисляются в списке через запятую. Например, инструкция

```
CREATE TABLE Personnel2
      (Code        INTEGER NOT NULL,
       Name        VARCHAR(30),
       PRIMARY KEY (Code));
```

создает таблицу Personnel2, для которой по столбцу Code строится первичный ключ. Отметим, что для ключевого столбца задан описатель NOT NULL.

Замечание

В связи с тем что в таблицах InterBase отсутствует такой тип, как автоинкрементный, в обязанности разработчика входит гарантия уникальности значений ключевых столбцов. Уникальность ключа обеспечивается на программном уровне через реализацию в приложении соответствующих алгоритмов выбора. Кроме того, можно использовать генераторы, которые будут рассмотрены ниже.

Построить первичный ключ можно также несколько иным способом, указав описатель `PRIMARY KEY` при определении соответствующего столбца:

```
CREATE TABLE Personnel2
     (Code     INTEGER NOT NULL PRIMARY KEY,
      Name     VARCHAR(30));
```

Как и в предыдущем примере, здесь создается таблица `Personnel2` с первичным ключом по столбцу `Code`. Оба приведенных способа определения ключа являются эквивалентными.

Таблица может иметь только один первичный ключ, однако кроме него можно определить еще несколько уникальных ключей. Для описания уникального ключа используется операнд `UNIQUE` следующего формата:

```
UNIQUE (<Список столбцов ключа>)
```

Так, инструкция:

```
CREATE TABLE Position
     (Code       INTEGER NOT NULL,
      Position   VARCHAR(20) NOT NULL,
      PRIMARY KEY (Code),
      UNIQUE (Position));
```

создает таблицу `Position`, для которой строятся два ключа — первичный ключ по столбцу `Code` и уникальный ключ по столбцу `Position`. Отметим, что таблица и один из ее столбцов могут иметь одинаковые имена (в данном случае это имя `Position`).

По своему назначению и использованию уникальный ключ не отличается от первичного и служит для обеспечения единственности значений ключевого столбца (столбцов) и ссылочной целостности.

Описание и использование внешнего ключа рассмотрено в следующем разделе.

Определение ограничений ссылочной целостности

Напомним, что действие ограничений ссылочной целостности заключается в следующем: если для записи главной таблицы имеются записи в подчиненной таблице (таблицах), то эту запись нельзя удалить, а также изменить значения столбцов, образующих ключ.

Определение ограничения ссылочной целостности имеет следующий формат:

```
[CONSTRAINT <Имя ограничения>]
FOREIGN KEY (<Список столбцов ключа>)
REFERENCES <Имя главной таблицы> [<Список столбцов ключа главной таблицы>]
```

При задании ограничений ссылочной целостности ключу (первичному или уникальному) главной таблицы ставится в соответствие внешний ключ подчиненной таблицы. Для описания внешнего ключа используется операнд FOREIGN KEY. Список этого операнда задает столбцы, по которым строится внешний ключ. Если внешний ключ используется для задания ограничений ссылочной целостности (для чего он собственно и предназначен), то его список столбцов должен соответствовать списку столбцов ключа главной таблицы.

В операнде REFERENCES указывается главная таблица. Описания столбцов ключа (первичного или уникального) главной таблицы и внешнего ключа подчиненной таблицы должны полностью совпадать.

Ограничения ссылочной целостности, как и другие ограничения, именуются. Имя может быть задано при определении ограничения в операнде CONSTRAINT, что, вообще говоря, не обязательно: если для ограничения ссылочной целостности не задано имя, то оно получает имя по умолчанию.

Пример задания ограничений ссылочной целостности:

```
CREATE TABLE Store
     (S_Code    INTEGER NOT NULL PRIMARY KEY,
      S_Name    VARCHAR(20) NOT NULL,
      S_Price   FLOAT,
      S_Number  Float);

CREATE TABLE Cards
     (C_Code      INTEGER NOT NULL PRIMARY KEY,
      C_Code2     INTEGER NOT NULL,
      C_Move      VARCHAR(20) NOT NULL,
      C_Date      DATE,
      CONSTRAINT  rStoreCards
FOREIGN KEY (C_Code2) REFERENCES Store);
```

Создаются две таблицы — Store и Cards, в каждой из которых определен первичный ключ. С таблицей Store связывается таблица Cards, для чего в ней по полю C_Code2 определен внешний ключ и задано ограничение ссылочной целостности с именем rStoreCards.

Замечание

При использовании программы IBConsole приведенные инструкции нужно вводить и выполнять поочередно или использовать файл сценария.

Аналогично можно определить для таблицы несколько различных ограничений ссылочной целостности.

Рассмотренные ограничения ссылочной целостности для связанных таблиц задаются на физическом уровне — уровне таблиц. Вместо этого на программном уровне можно использовать триггеры и реализовать с их помощью, например, каскадное удаление записей. В таком случае ограничения ссылочной целостности не определяются, и их требуется удалить, если они были ранее созданы.

Удалить ограничение ссылочной целостности можно следующей инструкцией:

```
ALTER TABLE <Имя таблицы>
      DROP <Имя ограничения ссылочной целостности>;
```

или из программы типа SQL Explorer.

Использование индексов

Создание и удаление индексов выполняется инструкциями `CREATE INDEX` и `DROP INDEX` соответственно. Инструкция создания индекса `CREATE INDEX` имеет формат:

```
CREATE [UNIQUE] {ASCENDING] [DESCENDING] INDEX
      <Имя индекса> ON <Имя таблицы> (<Имя столбца>, ..., [<Имя столбца>]);
```

В отличие от версии языка SQL для локальных БД, для индекса таблицы InterBase дополнительно можно задать несколько описателей:

- `UNIQUE` — индекс (как и ключ) требует уникальности значений столбца (столбцов), по которому он построен;
- `ASCENDING` — индексные столбцы сортируются в порядке возрастания значений (по умолчанию);
- `DESCENDING` — индексные столбцы сортируются в порядке убывания значений.

Так, инструкция

```
CREATE DESCENDING INDEX
       indNamePosition ON Personnel (Name, Position)
```

создает для таблицы `Personnel` индекс `indNamePosition`, построенный по полям `Name` и `Position`. Для индекса устанавливается сортировка в порядке убывания значений.

В остальном операции с индексами для таблиц InterBase и соответствующие средства языка SQL не отличаются от ранее рассмотренных для локальных БД.

Для каждого ключа таблицы автоматически строится соответствующий индекс, и ему присваивается имя по умолчанию. В отличие от обычного индекса, индекс, построенный по ключу, нельзя удалить инструкцией `DROP INDEX`: для этого нужно выполнить реструктуризацию таблицы инструкцией `ALTER TABLE`.

Описание индекса `indNamePosition` таблицы `Personnel`, созданного в предыдущем примере, имеет следующий вид:

```
INDNAMEPOSITION INDEX ON PERSONNEL (NAME, POSITION)
```

В свою очередь, строки

```
RDB$FOREIGN4 INDEX ON CARDS(C_CODE)
RDB$PRIMARY3 UNIQUE INDEX ON STORE(S_CODE)
```

описывают индексы для связанных таблиц, построенные на основании первичного ключа таблицы `STORE` и внешнего ключа таблицы `CARDS`. По умолчанию индексы получили имена `RDB$FOREIGN4` и `RDB$PRIMARY3` соответственно.

Приведенные описания получены с помощью программы IBConsole. Отметим, что просмотреть описания индексов можно и с помощью других программ, например, SQL Explorer, а также программно — с помощью инструкции

```
SHOW INDEX [<Имя таблицы>];
```

Эта инструкция выводит для указанной таблицы описания всех индексов, в том числе построенных по ключам. Если имя таблицы отсутствует, то выводятся описания всех индексов БД.

Как отмечалось, индекс обеспечивает быстрый доступ к данным, хранящимся в столбце (столбцах), для которого создан этот индекс. Это обеспечивается за счет использования индексно-последовательного метода доступа к данным.

При оценке эффективности (полезности) индекса применяется понятие глубины индекса. *Глубина индекса* представляет собой значение, показывающее максимальное число операций, которые необходимо выполнить при доступе к записи с использованием этого индекса. Допустимая глубина индекса не должна превышать двух операций, в противном случае говорят о разбалансировке индекса. *Разбалансировка индекса* может наступить при добавлении и удалении большого количества записей в БД. Отметим, что на глубину индекса также влияет размер страницы БД, который задается при ее создании. С увеличением размера страницы глубина индекса может уменьшиться.

Для уменьшения глубины индекса и устранения его разбалансировки нужно перестроить индекс путем выполнения следующей последовательности инструкций:

```
ALTER INDEX <Имя индекса> INACTIVE;
ALTER INDEX <Имя индекса> ACTIVE;
```

Первая инструкция деактивизирует (отключает) указанный индекс, а вторая — вновь активизирует (включает) его. Перестроить индекс можно и более кардинальным способом, удалив его и создав заново. Отметим, что вновь созданный индекс является активным.

Домены

Домен представляет собой именованное описание столбца. После определения домена его имя можно использовать при описании других столбцов. Домен удобен в случаях, когда несколько столбцов, в том числе принадлежащих различным таблицам, имеют одинаковое описание. Если провести сравнение с языком Object Pascal, то аналогом домена является тип данных. Таким образом, домен можно рассматривать как *шаблон* описания столбца.

Перед использованием домена его нужно создать с помощью инструкции

```
CREATE DOMAIN <Имя домена>
      [AS] <Описание домена>
```

Имя созданного домена затем можно включать в описания столбцов:

```
<Имя столбца> <Имя домена>
```

Описание домена не отличается от ранее рассмотренного описания столбца и может включать различные ограничения, применимые к столбцу. Однако в отличие от огра-

ничений для столбца, имя домена нельзя использовать в выражениях для ограничений: вместо имени домена в них указывается слово VALUE.

Пример использования доменов:

```
CREATE DOMAIN D_Position AS VARCHAR(20) NOT NULL;

CREATE DOMAIN D_Price AS FLOAT CHECK(VALUE > 0);

CREATE TABLE Position
      (Code        INTEGER NOT NULL PRIMARY KEY,
      Position     D_Position,
      Note         VARCHAR(50));

CREATE TABLE Personnel
      (Code       INTEGER NOT NULL PRIMARY KEY,
      Name        VARCHAR(30),
      Position    D_Position,
      Salary      FLOAT,
      BirthDay    DATE);
```

Домен D_Position представляет собой описание строкового столбца длиной не более 20 символов, который не может быть пустым. Этот домен использован при описании столбцов Position таблиц Position и Personnel. Второй домен D_Price представляет собой числовой столбец, значение которого должно быть положительным. Это ограничение задано конструкцией CHECK(VALUE > 0). Для приводимых таблиц домен D_Price не нужен, а создан с целью последующего использования.

Просмотры

Просмотр (view) является логической (виртуальной) таблицей, записи в которую отобраны с помощью инструкции SELECT. Преимущество просмотра заключается в том, что можно один раз отобрать записи и использовать их в дальнейшем без повторного выполнения инструкции SELECT. Это выгодно при частом выполнении одинаковых запросов, особенно тех, в которых реализованы сложные условия отбора. Иногда просмотр также называют *представлением*.

Просмотр создается следующей инструкцией:

```
CREATE VIEW <Имя просмотра> [<Список столбцов>]
      AS <Инструкция SELECT> [WITH CHECK OPTION];
```

Имя просмотра после его создания можно использовать как имя физической таблицы. Состав столбцов просмотра определяет список, который следует за именем просмотра. Если список столбцов не задан, то в просмотр отбираются все столбцы таблиц, указанных в инструкции SELECT. Инструкция SELECT составляется обычным образом.

Просмотр может быть редактируемым (модифицируемым) или нередактируемым. Для того чтобы записи просмотра можно было изменять, удалять и добавлять, должны выполняться как минимум следующие условия:

- в просмотр отобраны записи только из одной таблицы;
- в просмотр включены все столбцы таблицы, имеющие ограничение NOT NULL;

- в инструкции SELECT не были использованы статистические функции, хранимые процедуры, пользовательские функции, описатель DISTINCT, операнд HAVING, а также соединение таблиц.

Операнд WITH CHECK OPTION для редактируемого просмотра запрещает добавление записей, не удовлетворяющих условиям отбора, заданным в инструкции SELECT.

Для иллюстрации использования просмотра рассмотрим следующий пример:

```
CREATE VIEW vStore AS
      SELECT S_Name, C_Number
            FROM Store
            WHERE S_Number >= 0.5

SELECT * FROM vStore
```

Создается просмотр, состоящий из столбцов S_Name (название) и C_Number (количество товара) таблицы Store. В просмотр отбираются записи, для которых количество товара равно или превышает 0.5. Далее в инструкции SELECT просмотр vStore используется как обычная таблица.

Удалить просмотр можно следующей инструкцией:

```
DROP VIEW <Имя просмотра>;
```

Например:

```
DROP VIEW vStore;
```

Хранимые процедуры

Хранимая процедура представляет собой подпрограмму, расположенную на сервере и вызываемую из приложения клиента. Использование этих объектов увеличивает скорость доступа к БД по следующим причинам:

- вместо текста запроса, который может быть достаточно длинным, серверу передается по сети относительно короткое обращение к хранимой процедуре;
- хранимая процедура, в отличие от запроса, не требует предварительной синтаксической проверки.

Еще одним преимуществом при обращении к хранимым процедурам является то, что будучи общими для всех приложений-клиентов, они реализуют единые для них правила работы с БД.

Для выполнения процедуры, хранимой на сервере, в приложении используется компонент StoredProc.

Использование хранимых процедур

Хранимая процедура создается инструкцией

```
CREATE PROCEDURE
      <Имя процедуры> [(<Список входных параметров>)]
      [RETURNS (<Список выходных параметров>)]
      AS <Тело процедуры>
```

После имени процедуры идет необязательный список входных параметров, с помощью которых из приложения в процедуру передаются исходные данные. В свою очередь, список выходных параметров, с помощью которых в приложения передаются результаты выполнения процедуры, указывается после описателя RETURNS. Для каждого параметра указываются его имя и тип (через пробел), разделителем параметров в списке служит запятая.

```
<Список параметров> =
       [<Имя параметра1> <Тип параметра1>,]
       ...
       [<Имя параметраN> <Тип параметраN>]
```

При задании имен процедур и параметров (как и других объектов) для удобства целесообразно использовать мнемонические правила. Например, имя процедуры можно начинать с префикса p (от англ. *procedure*), имя входного параметра — с ip (от англ. *input parameter*), а имя выходного параметра — с op (от англ. *output parameter*).

При использовании параметра в выражениях тела процедуры перед его именем нужно указывать двоеточие (:).

Тело процедуры состоит из двух частей — описательной и исполнительной — и имеет следующий формат:

```
[<Объявление переменных>]
BEGIN
      <Инструкция>
      ...
      [<Инструкция>]
END
```

В описательной части объявляются переменные, используемые внутри процедуры. Эти переменные являются локальными и по окончании работы процедуры теряют свои значения. В исполнительной части располагаются инструкции, выполняющие необходимые действия. Как и в языке Pascal, инструкции располагаются между ключевыми словами BEGIN и END. В теле процедуры должна содержаться как минимум одна инструкция.

Созданную хранимую процедуру можно удалить или изменить. Удаляется процедура инструкцией

```
DROP PROCEDURE <Имя процедуры>
```

Изменение процедуры выполняется инструкцией ALTER PROCEDURE, имеющей такой же формат, как и инструкция CREATE PROCEDURE. После выполнения инструкции ALTER PROCEDURE предыдущее описание хранимой процедуры с указанным именем заменяется на новое описание.

Язык хранимых процедур

Для написания хранимых процедур и триггеров используется язык хранимых процедур. Язык хранимых процедур сервера InterBase представляет собой процедурный алгоритмический язык, а его синтаксис имеет много общего с синтаксисом языка Delphi.

Язык хранимых процедур включает в свой состав инструкции, позволяющие управлять ходом вычислительного процесса (условную инструкцию и инструкцию цикла). Кроме того, язык хранимых процедур обладает рядом возможностей языка SQL.

Рассмотрим кратко основные составляющие языка хранимых процедур.

В текст процедуры (триггера) можно вставлять комментарии. Для ограничения комментариев используются символы `/*` и `*/`.

В язык хранимых процедур включены инструкции:

- объявления переменных;
- присваивания;
- условные;
- составные;
- цикла;
- выбора записи;
- выбора нескольких записей (набора данных);
- возврата значений;
- выхода из процедуры;
- вызова процедуры;
- посылки сообщения.

Инструкции должны заканчиваться точкой с запятой (кроме составной инструкции).

Инструкция объявления переменных имеет следующий формат:

```
DECLARE VARIABLE <Имя переменной> <Тип переменной>;
```

Допустимыми типами для переменных являются типы столбцов InterBase, например, `VARCHAR` или `DATE`.

Переменные являются локальными и действуют с момента входа в процедуру до момента выхода из нее. Например, в командах

```
DECLARE VARIABLE Number INTEGER;
DECLARE VARIABLE NewString VARCHAR(10);
DECLARE VARIABLE WorkDate DATE;
```

объявляются три переменные `Number`, `NewString` и `WorkDate` целого типа, типа "строка" и "дата" соответственно.

Инструкция присваивания предназначена для установки значения переменной и имеет вид

```
<Имя переменной> = <Выражение>;
```

При выполнении этой инструкции вычисляется значение выражения и присваивается переменной, имя которой задано в левой части инструкции. Можно указывать имена локальных переменных и выходных параметров, перед именем выходного параметра двоеточие не ставится. Переменная и выражение должны иметь совместимые типы, в противном случае при выполнении процедуры возникает ошибка. В качестве операн-

дов выражения можно использовать значения, имена переменных и параметров, встроенные и пользовательские функции, а также вызовы генераторов.

Так, в примере

```
DECLARE VARIABLE n INTEGER;
DECLARE VARIABLE s VARCHAR(30);
...
n = 17;
s = UPPER(s);
```

целочисленной переменной `n` присваивается значение `17`, а символы строки `s` переводятся в верхний регистр.

Замечание

Если тип выражения не соответствует типу переменной, то ошибка возникает при выполнении хранимой процедуры, а не при ее создании, что нужно учитывать при отладке хранимых процедур.

Условная инструкция в общем случае позволяет организовать ветвление вычислительного процесса и имеет следующий формат:

```
IF (<Условие>) THEN <Инструкция1> [ELSE <Инструкция2>];
```

`Условие` представляет собой выражение логического типа. Условная инструкция выполняется так: если условие истинно (имеет значение `True`), то выполняется `Инструкция1`, в противном случае — `Инструкция2`. Обе указанные инструкции могут быть составными.

Условная инструкция может быть записана в сокращенной форме, когда слово `ELSE` и инструкция после него отсутствуют. В этом случае при невыполнении условия управление сразу передается на инструкцию, следующую за условной.

Составная инструкция представляет собой группу из произвольного числа любых инструкций, ограниченную операторными скобками `BEGIN` и `END`. Составная инструкция имеет следующий формат:

```
BEGIN <Инструкция1> ... <ИнструкцияN> END
```

Независимо от числа входящих в нее инструкций, составная инструкция воспринимается как единое целое и может располагаться в любом месте, где допускается наличие инструкции. Наиболее часто составная инструкция используется в условных инструкциях и инструкциях цикла. В конце составной инструкции точка с запятой не ставится. Составная инструкция объединяет также все инструкции тела процедуры или триггера.

Пример использования условной и составной инструкций:

```
IF (x < 0)
      THEN BEGIN n = 0; s = ""; END
      ELSE n = 10;
```

Если значение числовой переменной `x` меньше нуля, то значения переменных `n` и `s` сбрасываются. Соответствующие инструкции присваивания объединены в составную инструкцию. Если условие не выполняется, то числовой переменной `n` присваивается новое значение, а значение строковой переменной `s` остается без изменений.

Инструкция цикла (повтора) обеспечивает повтор группы инструкций при соблюдении некоторых условий. Инструкция цикла имеет следующий формат:

```
WHILE (<Условие>) DO <Инструкция>;
```

Инструкция, расположенная после слова DO, будет выполняться до тех пор, пока соблюдается указанное условие. Если в цикле требуется выполнить несколько инструкций, то их нужно объединить в составную инструкцию. Например:

```
S = 0;
n = 1;
WHILE (n <= 10) DO
      BEGIN
            s = s + n;
            n = n + 1;
      END
```

Подсчитывается сумма чисел от 1 до 10. Перебор чисел и накапливание суммы выполняется в цикле.

Отметим, что на практике для подсчета суммы чисел удобнее использовать не хранимую процедуру сервера, а процедуру, расположенную в модуле Delphi. Этот и некоторые следующие примеры приводятся в упрощенном виде с целью показать отдельные инструкции языка хранимых процедур.

Инструкция выбора записи представляет собой инструкцию SELECT, которая возвращает одну строку, поэтому ее называют также инструкцией выбора записи. Значения столбцов возвращаемой строки присваиваются указанным переменным или параметрам. При своем выполнении инструкция выбора записи (как и инструкция SELECT) отбирает несколько строк, но возвращает только одну.

Формат инструкции выбора записи, по сравнению с форматом инструкции SELECT языка SQL, дополнен операндом:

```
INTO :<Имя1>, ... [:<ИмяN>]
```

Имя после двоеточия указывает переменную или выходной параметр, которым должны быть присвоены значения столбцов записи, полученной в результате выполнения инструкции SELECT. Эта инструкция часто используется совместно со статистическими функциями, такими как SUM (сумма) или AVG (среднее значение).

Рассмотрим использование инструкции SELECT в хранимой процедуре на примере:

```
CREATE PROCEDURE pSalary
      RETURNS (opSum FLOAT, opAvg FLOAT)
      AS
      BEGIN
            SELECT SUM(Salary), AVG(Salary)
            FROM Personnel
            INTO :opSum, :opAvg;
      END
```

Создается хранимая процедура pSalary, в которой для сотрудников организации (таблица Personnel) подсчитываются общая сумма зарплаты и средняя зарплата. Полученные значения присваиваются выходным параметрам opSum и opAvg, которые возвраща-

ют полученные значения в вызывающую программу. Входных параметров у процедуры нет.

Инструкция выбора нескольких записей также представляет собой инструкцию SELECT, способную возвращать несколько записей (в частности, ни одной). Инструкция выбора нескольких записей имеет формат:

```
FOR <Инструкция выбора записи> DO <Инструкция>;
```

Инструкция выбора нескольких записей может задавать не только отбор, но и обработку записей. После отбора записей согласно инструкции SELECT для каждой из них выполняется инструкция, указанная после слова DO.

Пример хранимой процедуры с инструкцией выбора:

```
CREATE PROCEDURE pCountCode
      RETURNS (opSum INTEGER)
      AS
      DECLARE VARIABLE n INTEGER;
      DECLARE VARIABLE x INTEGER;
      BEGIN
            x = 0;
            FOR SELECT Code FROM List
                  INTO n
                  DO x = x + n;
            opSum = x;
      END
```

Создается хранимая процедура pCountCode, которая вычисляет сумму значений целочисленного столбца Code таблицы List. Инструкция SELECT обеспечивает отбор записей таблицы, а инструкция присваивания увеличивает значение суммы x на значение столбца Code очередной записи. После перебора всех записей результат возвращается через выходной параметр opSum.

Часто после слова DO указывается инструкция SUSPEND возврата значений, которая передает в вызывающее приложение или хранимую процедуру значения выходных параметров и имеет вид:

```
SUSPEND;
```

Обычно инструкция SUSPEND применяется в инструкции SELECT выбора нескольких записей хранимой процедуры. Совместное использование этих инструкций фактически обеспечивает возврат в приложение совокупности записей, представляющих собой набор данных. Поэтому инструкцию выбора нескольких записей называют также *инструкцией, возвращающей набор данных*.

Рассмотрим пример:

```
CREATE PROCEDURE pSalary2 (ipSalaryMin FLOAT, ipSalaryMax FLOAT)
      RETURNS (opName VARCHAR(20), opSalary FLOAT)
      AS
      BEGIN
            FOR SELECT Name, Salary
            FROM Personnel
            WHERE Salary >= :ipSalaryMin
```

```
            AND Salary <= :ipSalaryMax
                  INTO :opName, :opSalary
                  DO SUSPEND;
      END
```

Создается хранимая процедура `pSalary2`, которая возвращает набор данных. Совокупность записей этого набора образуют записи таблицы `pSalary2`, соответствующие сотрудникам организации, у которых оклад принадлежит заданному диапазону. Границы диапазона определяют входные параметры `ipSalaryMin` и `ipSalaryMax`. Набор данных содержит столбцы `Name` и `Salary`.

Замечание

Хранимую процедуру, обеспечивающую возврат набора данных, можно вызывать с помощью средств, позволяющих получить одну строку результатов. При этом будет обрабатываться только одна (первая) запись получаемого набора. Такими средствами являются инструкция `EXECUTE PROCEDURE` вызова процедуры и компонент `StoredProc`, позволяющий обратиться к процедуре, хранимой на сервере. Обычно их используют только при отладке хранимых процедур, возвращающих набор данных.

Инструкция выхода из процедуры `EXIT` прекращает выполнение хранимой процедуры и осуществляет передачу управления в вызывающую программу или процедуру. Если инструкция выхода отсутствует, то завершение процедуры и передача управления в вызывающую программу происходят после выполнения последней инструкции процедуры. Например, инструкция

```
CREATE PROCEDURE pDivide (ipA FLOAT, ipB FLOAT)
      RETURNS (opRes FLOAT)
      AS
      BEGIN
            IF (pB = 0) THEN BEGIN pRes = 0; EXIT; END
            opRes =: ipA / ipB;
      END
```

создает хранимую процедуру `pDivide`, выполняющую деление двух чисел. Если делитель равен нулю, то результат обнуляется, и выполнение процедуры заканчивается.

Инструкция `EXECUTE` предназначена для вызова из процедуры другой хранимой процедуры и имеет формат:

```
EXECUTE PROCEDURE <Имя процедуры> [(<Список входных параметров>)]
       [RETURNING_VALUES (<Список выходных параметров>)]
```

Эта инструкция вызывает хранимую процедуру с указанным именем, входными и выходными параметрами. В частном случае у вызываемой процедуры параметры могут отсутствовать. Так, инструкция:

```
CREATE PROCEDURE pTest
      RETURNS (opResult FLOAT)
      AS
      BEGIN
            EXECUTE PROCEDURE pDivide (100, 20)
                  RETURNING_VALUES (opResult);
      END
```

создает хранимую процедуру pTest, которая в свою очередь вызывает ранее созданную процедуру pDivide деления двух чисел. В качестве входных параметров для деления передаются числа 100 и 20. Результат, возвращаемый процедурой pDivide, присваивается выходному параметру opResult процедуры pTest.

Инструкция POST_EVENT предназначена для отправки сообщения приложениям, связанным с сервером. Сообщение посылается всем приложениям при возбуждении в сервере определенного события. Инструкция отправки сообщения имеет следующий формат:

```
POST_EVENT "<имя события>";
```

Например, инструкцией

```
POST_EVENT "evInsertStore";
```

приложениям, связанным с сервером, посылается сообщение evInsertStore.

Отметим, что для получения соответствующего сообщения приложение должно предварительно зарегистрироваться на сервере. Для регистрации приложения и обеспечения посылки им сообщений можно использовать компонент IBEvents со страницы **InterBase** Палитры компонентов.

Виды хранимых процедур

По числу строк, возвращаемых в качестве результата, можно выделить следующие виды хранимых процедур:

- возвращающие одну строку;
- возвращающие несколько строк.

Процедуры, возвращающие одну строку, практически не отличаются от процедур языка Pascal и обеспечивают возврат значений выходных параметров. Такие хранимые процедуры также называют *процедурами действия*.

Процедуры, возвращающие несколько строк, передают набор данных, записями в котором являются строки результатов. Такие хранимые процедуры также называют *процедурами выбора*. В их теле размещаются совместно используемые инструкции выбора нескольких записей и возврата значений — FOR SELECT...DO...SUSPEND, которые и обеспечивают отбор требуемых записей и построчную передачу значений их столбцов в точку вызова.

Вызов хранимой процедуры представляет собой обращение к процедуре с передачей исходных данных для обработки и последующим получением результатов. Исходные данные передаются через входные параметры, а результаты возвращаются через выходные параметры. В зависимости от вида хранимой процедуры различаются способы ее вызова.

Вызов хранимой процедуры выбора

Хранимая процедура выбора возвращает набор данных, состоящий в общем случае из нескольких записей. Поэтому при ее вызове нужно обеспечить возможность приема совокупности записей. В связи с этим хранимая процедура выбора, как правило, вызы-

вается с помощью инструкции выбора SELECT, внутри которой размещается вызов процедуры. Например:

```
SELECT * FROM pSalary2 (2000, 3000)
```

Здесь вызывается хранимая процедура pSalary2, возвращающая фамилии и оклады сотрудников, значение оклада которых принадлежит заданному диапазону. Результат выполнения приведенной инструкции SELECT имеет следующий вид:

```
OPNAME          OPSALARY
=============== =============
Иванов П.Л.     2000
Петров О.Д.     2700
Кузнецов В.Ф.   2400
Васина Ю.Б.     2200
```

Замечание

Хранимую процедуру выбора в программе IBConsole также можно вызвать инструкцией EXECUTE PROCEDURE, например, так:

```
EXECUTE PROCEDURE pSalary2(2000, 3000)
```

В этом случае будет получена только первая запись набора данных. Напомним, что на практике такая инструкция вызова процедуры обычно используется только при отладке хранимых процедур выбора.

В приложении инструкция SELECT, вызывающая хранимую процедуру выбора, выполняется с помощью набора данных Query, для чего используется его свойство SQL.

Рассмотрим пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.SQL.Clear;
  Query1.SQL.Add('SELECT * FROM pSalary2 (2000, 3000)');
  Query1.Open;
end;
```

При нажатии кнопки Button1 свойству SQL компонента Query1 присваивается код вызова хранимой процедуры pSalary2, затем запрос выполняется. В результате набор данных Query1 содержит записи, удовлетворяющие заданному в процедуре условию отбора. Отметим, что столбцы набора данных образуют выходные параметры процедуры, в данном случае ими являются столбцы opName и opSalary.

Замечание

Для соединения с удаленной БД в форме, кроме набора данных Query, следует разместить компонент Database. (Вопросы соединения с удаленной БД из приложения были рассмотрены в *главе 25*.)

В приведенном примере SQL-запрос является статическим, т. к. входные параметры, ограничивающие диапазон оклада, передаются процедуре через явно указанные значения 2000 и 3000. Чтобы менять границы диапазона, можно считывать их значения, скажем, из компонентов Edit (полей редактирования).

Например:

```
Query1.SQL.Add('SELECT * FROM pSalary2 (' + Edit1.Text + ',' +
                                            Edit2.Text + ')');
```

Значения нижней и верхней границ диапазона для оклада вводятся в поля редактирования `Edit1` и `Edit2` соответственно.

Значения входных параметров процедуры также можно передавать через параметры компонента `Query1` (свойство `Params`). При этом запрос компонента `Query1` будет иметь вид:

```
'SELECT * FROM pSalary2 (:pMin, :pMax)'
```

Здесь границы диапазона для оклада задаются двумя параметрами: `pMin` — для нижней, а `pMax` — для верхней границы. Эти параметры являются входными и предназначены для передачи вещественных значений. Поэтому для них нужно задать следующие значения свойств:

- `DataType` (тип данных) — `ftFloat`;
- `ParamType` (тип параметра) — `ptInput`.

Управление свойствами параметров осуществляется с помощью Инспектора объектов и Редактора параметров SQL-запроса (рис. 26.2), который вызывается щелчком в области значения свойства `Params` в окне Инспектора объектов. При необходимости для параметров можно задать значения (свойство `Value`).

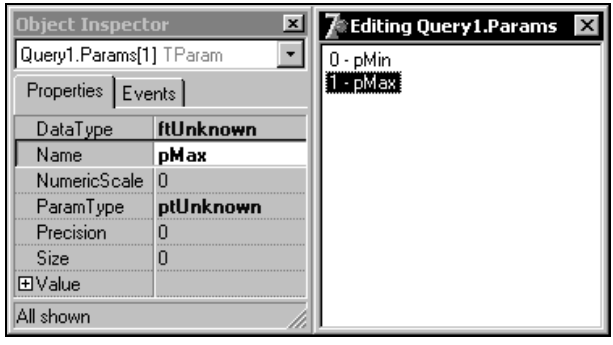


Рис. 26.2. Редактирование параметров SQL-запроса

При выполнении приложения значения параметров можно установить так:

```
Query1.ParamByName('pMin').Value := 2000;
Query1.ParamByName('pMax').Value := 3000;
```

или

```
Query1.ParamByName('pMin').Value := StrToFloat(Edit1.Text);
Query1.ParamByName('pMax').Value := StrToFloat(Edit2.Text);
```

Более подробно использование параметров SQL-запроса было рассмотрено в *главе 20*, посвященной реляционному способу доступа.

Вызов хранимой процедуры действия

Хранимая процедура действия вызывается инструкцией следующего формата:

```
EXECUTE PROCEDURE <Имя процедуры> [(<Список входных параметров>)]
```

Инструкция вызывает указанную процедуру, передавая ей значения входных параметров. Данная инструкция используется в программе IBConsole, при этом результаты работы процедуры (значения выходных параметров) выводятся в окне результатов выполнения запроса.

Например:

```
EXECUTE PROCEDURE pSalary
```

Ранее была создана хранимая процедура `pSalary`, в которой для сотрудников организации подсчитывались общая сумма зарплаты и средняя зарплата. Процедура не имеет входных параметров. В качестве выходных параметров процедуры заданы параметры `opSum` и `opAvg` для суммы и среднего значения соответственно. При выполнении инструкции `EXECUTE PROCEDURE` в среде программы IBConsole результат будет иметь следующий вид:

```
      OPSUM        OPAVG
=============== ================
      10300        2600
```

В программе IBConsole с помощью инструкции `EXECUTE PROCEDURE` также можно отлаживать хранимые процедуры выбора. Однако в этом случае в качестве их результатов возвращается только первая строка (первая запись набора данных).

Для вызова хранимой процедуры действия из приложения предназначен компонент `StoredProc`. Рассмотрим свойства и особенности использования этого компонента.

Свойство `DatabaseName` типа `String` указывает на компонент `Database`, используемый для соединения с БД. Соединение с удаленной БД компонента `StoredProc` не отличается от соединения с ней компонента `Query`, рассмотренного в *главе 25*.

Свойство `StoredProcName` типа `String` определяет вызываемую хранимую процедуру. Имя процедуры выбирается через список Инспектора объектов. Если при попытке выбрать процедуру соединение с БД еще не установлено (свойство `Connected` компонента `Database` имеет значение `False`), то выдается запрос на его установление. После ввода имени и пароля пользователя происходит соединение с БД, а свойство `Connected` компонента `Database` устанавливается в значение `True`.

Когда имя хранимой процедуры задано, становятся доступными ее входные и выходные параметры, определяемые значением свойства `Params` типа `TParams`. Это свойство определяет коллекцию (массив) параметров компонента `StoredProc`, работа с которыми аналогична работе с параметрами SQL-запроса компонента `Query`.

Свойство `ParamBindMode` типа `TParamBindMode` определяет, каким образом устанавливается соответствие между параметрами компонента `StoredProc` и параметрами процедуры. Оно может принимать следующие значения:

- `pbByName` (соответствие устанавливается по именам) — по умолчанию; имена параметров компонента `StoredProc` и соответствующих параметров процедуры должны совпадать;

- pbByNumber (соответствие устанавливается по номерам); первый параметр компонента StoredProc соотносится с первым параметром процедуры, второй — со вторым и т. д.

Выполнение выбранной хранимой процедуры осуществляется последовательным вызовом методов Prepare и ExecProc. Метод Prepare подготавливает хранимую процедуру к выполнению путем связывания параметров компонента StoredProc и параметров процедуры. Метод ExecProc выполняет процедуру.

При необходимости перед выполнением процедуры можно установить или изменить значения ее входных параметров аналогично тому, как это делается для параметров SQL-запроса. Результаты работы процедуры возвращаются в выходных параметрах. Для доступа к параметрам при выполнении приложения удобно использовать метод ParamByName.

Рассмотрим пример, иллюстрирующий вызов хранимой процедуры действия:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  StoredProc1.StoredProcName := 'pSalary';
  StoredProc1.Prepare;
  StoredProc1.ExecProc;
  Edit1.Text := StoredProc1.ParamByName('opSum').Value;
  Edit2.Text := StoredProc1.ParamByName('opAvg').Value;
end;
```

Как и в предыдущем примере, здесь вызывается процедура действия pSalary. Ее результаты — значения выходных параметров opSum и opAvg — выводятся в компонентах Edit1 и Edit2 соответственно. Для вызова процедуры используется компонент StoredProc1.

Триггеры

Триггер представляет собой процедуру, которая находится на сервере БД и вызывается автоматически при модификации записей БД, т. е. при изменении столбцов или при их удалении и добавлении. В отличие от хранимых процедур, триггеры нельзя вызывать из приложения клиента, а также передавать им параметры и получать от них результаты.

Триггер по своей сути похож на обработчики событий BeforeEdit, AfterEdit, BeforeInsert, AfterInsert, BeforeDelete и AfterDelete, связанных с модификацией таблиц. Триггер может вызываться при редактировании, добавлении или удалении записей до и/или после этих событий.

Замечание

Изменения, внесенные триггером в транзакции, которая оказалась отмененной, также отменяются.

Триггеры обычно используются для программной реализации так называемых бизнес-правил. С их помощью удобно реализовывать различные ограничения, например, ограничения на значения столбцов или ограничения ссылочной целостности, а также вы-

полнять такие действия, как накапливание статистики работы с БД или резервное копирование записей.

Замечание

Устанавливаемые триггером бизнес-правила не должны противоречить аналогичным правилам, заданным для БД на физическом уровне.

Создание и изменение триггера

Создание триггера выполняется инструкцией CREATE TRIGGER, имеющей формат:

```
CREATE TRIGGER <Имя триггера> FOR <Имя таблицы>
      [ACTIVE | INACTIVE]
      {BEFORE | AFTER}
      {UPDATE | INSERT | DELETE}
       [POSITION <Число>]
            AS <Тело триггера>
```

Описатели ACTIVE и INACTIVE определяют активность триггера сразу после его создания. По умолчанию действует ACTIVE, и созданный триггер активен, т. е. при наступлении соответствующего события будет выполняться. Если триггер неактивен, то при наступлении соответствующего события он не вызывается. Ранее созданный триггер можно активизировать или, наоборот, деактивизировать.

Описатели BEFORE и AFTER задают момент начала выполнения триггера до или после наступления соответствующего события, связанного с изменением записей.

Описатели UPDATE, INSERT и DELETE определяют, при наступлении какого события вызывается триггер — при редактировании, добавлении или удалении записей соответственно.

Для одного события можно создать несколько триггеров, каждый из которых будет автоматически выполнен (если находится в активном состоянии). При наличии нескольких триггеров порядок их вызова (выполнения) определяет число, указанное в операнде POSITION. Триггеры выполняются в порядке возрастания этих чисел.

Созданный триггер можно удалить или изменить. Удаляется триггер инструкцией

```
DROP TRIGGER <Имя триггера>
```

Изменение триггера выполняется инструкцией ALTER TRIGGER, формат которой не отличается от формата инструкции создания триггера. После выполнения инструкции ALTER TRIGGER предыдущее описание триггера с указанным именем заменяется на новое.

Программирование триггера аналогично программированию хранимой процедуры, для чего используется специальный язык, позволяющий также создавать хранимые процедуры.

Отметим, что для доступа к значениям столбца используются конструкции формата:

```
OLD.<Имя столбца>
NEW.<Имя столбца>
```

Первая из них позволяет обратиться к старому (до внесения изменений), а вторая — к новому (после внесения изменений) значениям столбца.

Примеры использования триггера

Рассмотрим использование триггера для реализации ограничений ссылочной целостности и для занесения в ключевые столбцы уникальных значений.

В связи с тем, что InterBase не поддерживает автоинкрементные поля, при создании ключевого столбца, требующего уникальности значений, рекомендуется поступать так:

1. При создании таблицы задать ключевой столбец целочисленного типа.
2. Создать генератор, который при обращении к нему возвращает уникальное целочисленное значение.
3. Создать триггер, который при добавлении к таблице новой записи обращается к генератору и заносит возвращаемое им значение в ключевое поле.

Следующий пример иллюстрирует описанную последовательность действий:

```
/* Создание таблицы */
CREATE TABLE Store
     (S_Code  INTEGER NOT NULL,
      ...
      PRIMARY KEY (S_Code));

/* Создание генератора */
CREATE GENERATOR GenStore;
SET GENERATOR GenStore TO 1;

/* Создание триггера */
CREATE TRIGGER CodeStore FOR Store
ACTIVE
BEFORE INSERT
      AS
      BEGIN
            NEW.S_Code = GEN_ID(GenStore, 1);
      END
```

После добавления к таблице `Store` новой записи ключевому столбцу `S_Code` этой записи автоматически присваивается уникальное значение. Это обеспечивается обращением `GEN_ID` к генератору `GenStore`, который создается отдельно от триггера (работа с генераторами рассмотрена ниже).

Напомним, что ограничения ссылочной целостности для связанных таблиц включают в себя:

- каскадное удаление записей;
- запрет на редактирование ключевых столбцов.

Рассмотрим, как можно реализовать каскадное удаление записей с участием триггера.

Пусть имеются две таблицы: главная `Store` и подчиненная `Cards`, связанные по полям кода `S_Code` и `C_Code2` соответственно:

```
CREATE TABLE Store
     (S_Code      INTEGER NOT NULL,
      ...
      PRIMARY KEY (S_Code));
```

```
CREATE TABLE Cards
     (C_Code      INTEGER NOT NULL,
      C_Code2     INTEGER NOT NULL,
      ...
      PRIMARY KEY (C_Code));
```

Напомним, что каскадное удаление записей для связанных таблиц заключается в том, что если из главной таблицы удаляется запись, то и в подчиненной таблице должны быть удалены все соответствующие ей записи.

В нашем случае это выполняется следующим образом:

```
CREATE TRIGGER DeleteStore FOR Store
ACTIVE
AFTER DELETE
      AS
      BEGIN
            DELETE FROM Cards WHERE Store.S_Code = Cards.C_Code2;
      END
```

После удаления записи в таблице `Store` (склад) будут автоматически удалены все соответствующие записи в таблице `Cards` (движение товара).

Замечание

Для таблиц не должны действовать ограничения ссылочной целостности, заданные на физическом уровне, например, так:

```
CONSTRAINT rStoreCards
FOREIGN KEY (C_Code2) REFERENCES Store
```

В противном случае при попытке удалить запись из главной таблицы генерируется исключение. Если есть такие ограничения, то их можно удалить, например, с помощью программы SQL Explorer.

Аналогичным способом можно реализовать и обновление столбцов связи (ключевых столбцов) связанных таблиц, заключающееся в том, что при изменении значения столбца связи главной таблицы соответственно изменяются значения столбца связи всех связанных записей подчиненной таблицы. Например:

```
CREATE TRIGGER ChangeStore FOR Store
ACTIVE
BEFORE UPDATE
      AS
      BEGIN
            IF (OLD.S_Code <> NEW.S_Code)
                  THEN UPDATE Cards
                        SET C_Code2 = NEW.S_Code
                        WHERE C_Code2 = OLD.S_Code;
      END
```

При изменении столбца `S_Code`, используемого для связи главной таблицы `Store` с подчиненной таблицей `Cards`, автоматически изменяются значения столбца связи `C_Code2` соответствующих записей подчиненной таблицы.

Замечание

Чтобы столбец связи главной таблицы можно было редактировать, по нему не должен быть создан ключ. Если ключ создан, то его следует удалить, например, с помощью программы SQL Explorer. Более кардинальным решением проблемы является создание таблицы заново с помощью инструкции `ALTER TABLE`.

Поскольку использование триггеров не допускает существования ограничений на физическом уровне (при определении структуры таблиц), необходимо обеспечить также занесение значения в столбец связи подчиненной таблицы при добавлении в нее новой записи.

Создание генераторов

Напомним, что, в отличие от базы данных Paradox, для таблиц InterBase отсутствует автоинкрементный тип, обеспечивающий автоматическую установку уникальных значений. Поэтому для обеспечения уникальности значений ключевых столбцов совместно с триггерами используются генераторы. *Генератор* возвращает уникальное целочисленное значение.

С помощью языка SQL-сервера можно создать генератор и установить для него начальное значение. Генератор создается следующей инструкцией:

```
CREATE GENERATOR <Имя генератора>;
```

Начальное значение задается инструкцией:

```
SET GENERATOR <Имя генератора> TO <Начальное значение>;
```

Начальное значение представляет собой целое число, начиная с которого формируется числовой ряд.

Обращение к созданному генератору выполняется с помощью функции

```
GEN_ID(<Имя генератора>, <Шаг>);
```

Эта функция возвращает значение, увеличенное на целочисленный шаг относительно предыдущего значения генератора.

Замечание

После определения начального значения генератора изменять его нельзя. Также нельзя изменять шаг. Невыполнение этого правила приведет к тому, что генератор может повторно возвратить уже имеющееся значение со всеми вытекающими последствиями.

Пример создания генератора:

```
CREATE GENERATOR GenStore;
SET GENERATOR GenStore TO 1;
```

Здесь создается генератор `GenStore`, имеющий начальное значение `1`. Пример обращения к этому генератору приведен в предыдущем разделе — `GEN_ID(GenStore, 1)`.

Функции, определяемые пользователем

Функция, определяемая пользователем, представляет собой обычную функцию, написанную на алгоритмическом языке, например, Pascal. Созданная функция оформляется в виде динамической библиотеки DLL, откуда может быть вызвана обычным способом. В общем случае библиотека содержит несколько функций. Для обеспечения вызова функции системе Windows должен быть известен путь к соответствующей библиотеке.

Достоинства применения функций, определяемых пользователем, состоят в следующем:

- расширяется состав функций языка SQL;
- появляется возможность использования функций другими приложениями.

Отметим, что состав встроенных функций языка SQL относительно небольшой, и в нем отсутствуют, например, функции обработки дат, времени и чисел, а для строк реализованы только несколько наиболее употребительных операций. При необходимости пользователь может сам создать и в дальнейшем использовать такие часто применяемые функции, как вычисление модуля или квадратного корня.

Порядок действий при работе с пользовательской функцией следующий:

1. Создать функцию и включить ее в соответствующую библиотеку; при необходимости создается также и библиотека (модуль DLL).
2. Объявить функцию в сервере.
3. Вызвать функцию в инструкции SQL.

Рассмотрим в качестве примера использование функции, выполняющей извлечение квадратного корня из числа. Назовем функцию `Sqrt2`, а библиотеку `Ibdll`.

В среде Delphi код библиотеки имеет следующий вид:

```
LIBRARY Ibdll;

USES
      SysUtils;
      Classes;

FUNCTION Sqrt2 (x: REAL): REAL; CDECL; EXPORT;
BEGIN
  IF x < 0 THEN Sqrt2 := 0 ELSE Sqrt2 := SQRT(x);
END;

EXPORTS
  Sqrt2;

BEGIN
END.
```

По сравнению с обычным для модулей Delphi описанием функции в ее заголовке указаны слова CDECL и EXPORT, определяющие соглашения для параметров и назначение функции. Имя экспортируемой функции указывается в разделе EXPORTS.

Полученный в результате трансляции модуль библиотеки Ibdll.dll нужно скопировать в каталог, где этот модуль будет доступен системе Windows, например, в каталог C:\WINNT\SYSTEM32.

Объявление пользовательской функции в сервере выполняется с помощью следующей инструкции:

```
DECLARE EXTERNAL FUNCTION <Имя функции>
      [<Тип данных1>, ... <Тип данныхN>]
      RETURNS {<Тип данных> [BY VALUE] | CSTRING (<Целое число>)}
      ENTRY_POINT "<Зарегистрированное имя функции>"
      MODULE_NAME "<Имя библиотеки>";
```

Имя функции указывает имя, с помощью которого ее можно вызывать из сервера. В общем случае это имя не совпадает с именем, под которым функция регистрируется в библиотеке при ее создании (операнд ENTRY_POINT). За именем следует список типов входных параметров функции. Отметим, что строковые значения должны передаваться по ссылке (указателю). Поэтому для строковых параметров следует указывать тип CSTRING, который соответствует типу PChar языка Pascal. Описание типа CSTRING имеет формат:

```
CSTRING (<Число символов строки>)
```

Тип возвращаемого функцией результата указывается после слова RETURNS. По умолчанию результат возвращается по ссылке, при необходимости обеспечить его возврат по значению указывается описатель BY VALUE. Имя используемой библиотеки определяет операнд MODULE_NAME, в котором задается имя модуля библиотеки DLL (без расширения).

Регистрацию рассмотренной функции выполняет инструкция

```
DECLARE EXTERNAL FUNCTION Sqrt2 FLOAT
      RETURNS FLOAT BY VALUE
      ENTRY_POINT "Sqrt2"
      MODULE_NAME "IBDLL";
```

Отмену объявления функции в сервере (удаление) выполняет инструкция:

```
DROP EXTERNAL FUNCTION <Имя функции>;
```

После объявления пользовательской функции ее можно вызывать в инструкциях языка SQL наряду с его встроенными функциями, такими как SUM, AVG или CAST. Например, вызов функции Sqrt2 в инструкции отбора записей будет иметь следующий вид:

```
SELECT * FROM Graphics
      WHERE CoordY <= Sqrt2(CoordX)
```

Приведенная инструкция отбирает из таблицы Graphics записи, для которых значение столбца CoordY не превосходит квадратного корня из значения столбца CoordX.

Отметим, что локальная версия сервера InterBase не поддерживает работу с функциями, определяемыми пользователем.

Реализация механизма транзакций

В *главе 19*, посвященной навигационному способу доступа с помощью BDE, механизм транзакций был рассмотрен на примере локальных БД. Напомним, что механизм транзакций используется для поддержания целостности БД: транзакция переводит БД из одного целостного состояния в другое. Чтобы транзакция была успешной, должны выполниться все операции, входящие в ее состав. В случае возникновения ошибки хотя бы в одной из операций вся транзакция считается неуспешной, и результаты всех операций отменяются. Транзакция может быть явной и неявной.

Неявная транзакция запускается и завершается автоматически, *явной* транзакцией управляет программист. С явным управлением транзакциями для локальных БД с помощью методов и свойств компонента `Database` мы познакомились в *главе 19*. Рассмотренные средства компонента `Database` можно использовать и для удаленных баз данных. Кроме того, для удаленных БД имеются дополнительные возможности по управлению транзакциями.

При модификации удаленной БД также осуществляются неявные и явные транзакции. Для удаленных БД, кроме поддержания целостности данных, использование механизма транзакций позволяет определить порядок взаимодействия запросов. Для соперничающих запросов устанавливается режим одновременного доступа к одним и тем же данным.

Для модификации данных может использоваться SQL-запрос, выполняемый с помощью метода `ExecSQL` компонента `Query`. Такой запрос называют PassThrough SQL, его выполнение приводит к запуску неявной транзакции. Способ взаимодействия с сервером на уровне такой транзакции определяет параметр `SQLPASSTHRU MODE` псевдонима БД или драйвера (в нашем случае типа InterBase), который может принимать следующие значения:

- `SHARED AUTOCOMMIT` — инструкциями модификации БД, например, `UPDATE` или `INSERT`, автоматически запускается неявная транзакция (по умолчанию); после внесения изменений эта транзакция автоматически подтверждается; разные транзакции могут использовать общее соединение с БД;
- `SHARED NO AUTOCOMMIT` — инструкциями модификации БД также автоматически запускается неявная транзакция, но ее автоматического подтверждения не происходит, и нужно самостоятельно выполнять инструкцию `COMMIT`; разные транзакции могут использовать общее соединение с БД;
- `NOT SHARED` — транзакции должны использовать различные соединения с БД и подтверждаться выполнением инструкции `COMMIT`.

Возможность явного управления транзакциями предоставляет язык SQL сервера, в составе которого есть следующие инструкции:

- `SET TRANSACTION` (начать транзакцию);
- `COMMIT` (подтвердить транзакцию);
- `ROLLBACK` (откатить транзакцию).

Инструкция запуска явной транзакции имеет формат:

```
SET TRANSACTION
[READ WRITE | READ ONLY]
[WAIT | NO WAIT]
[[ISOLATION LEVEL]
      {SNAPSHOT [TABLE STABILITY] | READ COMMITED}]
[RESERVING <Список таблиц>
      [FOR [{SHARED | PROTECTED}] [{READ | WRITE}]];
```

Все операнды этой инструкции являются необязательными и позволяют управлять перечисленными ниже режимами транзакции.

- Режим доступа к данным:
  - `READ WRITE` (разрешены чтение и модификация записей) — по умолчанию;
  - `READ ONLY` (разрешено только чтение записей).
- Поведение в случае конфликта транзакций при обновлении записей:
  - `WAIT` (ожидание завершения другой транзакции) — по умолчанию;
  - `NO WAIT` (прекращение данной транзакции).
- Уровень изоляции от других транзакций (операнд `ISOLATION LEVEL`):
  - `SNAPSHOT` (чтение данных в состоянии на момент начала транзакции) — по умолчанию; изменения, сделанные другими транзакциями, в данной транзакции не видны;
  - `SNAPSHOT TABLE STABILITY` (предоставление транзакции исключительного доступа к таблицам); другие транзакции могут читать записи из таблиц;
  - `READ COMMITED` (чтение только подтвержденных изменений в записях); если изменения еще не подтверждены, то читается предыдущая версия записи.
- Блокирование таблиц, указанных в списке операнда `RESERVING`, для других транзакций:
  - `PROTECTED READ` (разрешено только чтение записей);
  - `PROTECTED WRITE` (для транзакций с уровнем изоляции `SNAPSHOT` или `READ COMMITTED` разрешено только чтение записей);
  - `SHARED READ` (разрешены чтение и модификация записей);
  - `SHARED WRITE` (разрешено чтение записей; для транзакций с уровнем изоляции `SNAPSHOT` или `READ COMMITTED` разрешена модификация записей).

Действие инструкций `COMMIT` и `ROLLBACK` по подтверждению и откату транзакции аналогично действию одноименных методов компонента `Database`.

Механизм кэшированных изменений

Механизм кэшированных изменений заключается в том, что на компьютере клиента в кэше (буфере) создается локальная копия данных, и все изменения в данных выполня-

ются в этой копии. Для хранения локальной копии используется специальный буфер (кэш). Сделанные изменения можно подтвердить, перенеся их в основную БД, хранящуюся на сервере, или отказаться от них. Этот механизм напоминает механизм транзакций, но, в отличие от него, снижает нагрузку на сеть, т. к. все изменения передаются в основную БД одним пакетом. Следует иметь в виду, что для всех записей локальной копии отсутствуют блокировки на изменение их значений. Блокировки могут быть установлены другими приложениями для основной БД, находящейся на сервере.

Механизм кэшированных изменений реализуется в приложении, для чего компоненты, в первую очередь `Database`, `Table` и `Query` (используемые при доступе с помощью BDE), имеют соответствующие средства. Кроме того, механизм кэшированных изменений поддерживается предназначенным для этого компонентом `UpdateSQL`.

Основные достоинства рассматриваемого механизма проявляются для удаленных БД, но его можно использовать и при работе с локальными БД.

Компоненты *Database*, *Table* и *Query*

Включением для набора данных режима кэшированных изменений управляет свойство `CachedUpdates` типа `Boolean`: значение `True` этого свойства включает (активизирует) режим, значение `False` — выключает его (по умолчанию). Для рассматриваемых в этой главе удаленных БД и механизма доступа BDE, как правило, используется набор данных `Query`, в то время как для локальных БД нет принципиальной разницы между использованием наборов данных `Table` и `Query`.

После включения режима кэшированных изменений данные автоматически копируются в буфер и дальнейшая работа осуществляется именно с этими данными. Остальные пользователи (наборы данных) не видят проделанных изменений до тех пор, пока они не перенесены в основную БД. Для того чтобы измененные данные из буфера были переписаны в основную БД, нужно подтвердить изменения. Подтверждение кэшированных изменений выполняется в два этапа:

- запись кэшированных изменений в основную БД;
- подтверждение или отмена сделанных изменений.

Приведенный порядок действий напоминает порядок действий при управлении транзакцией. Для их реализации нужно вызвать соответствующие методы набора данных или компонента `Database`, с которым связан набор.

Метод `ApplyUpdates` набора данных записывает в БД изменения в кэшированных данных. При этом результаты выполнения всех операций редактирования, вставки или удаления записей пересылаются одним пакетом, что может значительно снизить нагрузку на сеть. Вызов метода `ApplyUpdates` производится на первом этапе. При записи данных в основную БД возможен вариант, когда сделанные изменения вызовут исключение, например, связанное с нарушениями ограничений ссылочной целостности или блокировкой записей другими пользователями. Поэтому необходимо перехватывать и обрабатывать возможное исключение. Для анализа результата выполнения первого этапа удобно использовать конструкцию `TRY...EXCEPT` локальной обработки исключений.

На втором этапе в зависимости от результата первого этапа сделанные изменения утверждаются или отменяются. Метод `CommitUpdates` набора данных подтверждает изменения, а метод `CancelUpdates` отменяет их, возвращая БД в исходное состояние.

Методы, реализующие оба этапа подтверждения кэшированных изменений, рекомендуется выполнять в рамках транзакции, чтобы обеспечить возможность восстановления данных при возникновении ошибок.

Так, в процедуре

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Database1.StartTransaction;
  try
    Query1.ApplyUpdates;
    Database1.Commit;
    Query1.CommitUpdates;
  except
    MessageDlg('Изменения не сохранены!', mtError, [mbOK], 0);
    Query1.CancelUpdates;
  end;
end;
```

выполняется подтверждение кэшированных изменений, сделанных в наборе данных `Query1`. Перед выполнением этой операции запускается явная транзакция, которая при отсутствии ошибок также явно подтверждается. При возникновении ошибки, связанной с выполнением любого из методов, необходимых для подтверждения изменений, эти изменения отменяются, о чем пользователю выдается сообщение. Проверку успешности операций подтверждения осуществляет локальный обработчик исключений.

Метод `ApplyUpdates(const DataSets: array of TDBDataSet)` компонента `Database` используется для подтверждения изменений одновременно в нескольких наборах данных, заданных в параметре `DataSets`. В отличие от одноименного метода набора данных, метод `ApplyUpdates` компонента `Database` при записи изменений в БД автоматически запускает транзакцию и при ее успешном завершении также автоматически подтверждает сделанные изменения. Таким образом, при отсутствии ошибок этот метод реализует оба этапа выполнения транзакции. В случае возникновения ошибки необходимо самостоятельно вызывать метод `CancelUpdates` набора данных, отменяющий изменения.

Замечание

Если набор данных не находится в режиме кэшированных изменений и его свойство `CachedUpdates` имеет значение `False`, то вызов методов `ApplyUpdates`, `CommitUpdates` или `CancelUpdates` приводит к исключению.

Пример подтверждения кэшированных изменений в рамках неявной транзакции:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  try
    Database1.ApplyUpdates([Query1]);
```

```
  except
    MessageDlg('Изменения не сохранены!', mtError, [mbOK], 0);
    Query1.CancelUpdates;
  end;
end;
```

Как и в предыдущем примере, выполняется подтверждение кэшированных изменений, сделанных в наборе данных `Query1`. При возникновении ошибок подтверждения сделанные изменения отменяются.

В приведенных примерах кэшированные изменения или целиком подтверждались, или целиком отменялись. Кроме этого, при возникновении ошибок подтверждения изменений возможен анализ отдельных записей и их столбцов (полей), в результате чего выявляются приведшие к ошибке записи и корректируются неправильные данные. Такую возможность предоставляет событие `OnUpdateError` типа `TUpdateErrorEvent`, в обработчике которого следует выполнить действия по исправлению ситуации. Тип события описан следующим образом:

```
type TUpdateErrorEvent =
   procedure(DataSet: TDataSet; E: EDatabaseError; UpdateKind:
             TUpdateKind; var UpdateAction: TUpdateAction) of object;
```

Список параметров, передаваемых обработчику, приведен далее:

- `DataSet` — набор данных, при сохранении изменений которого произошла ошибка;
- `UpdateKind` — вид изменений записи:
  - `ukModify` (модификация);
  - `ukInsert` (вставка);
  - `ukDelete` (удаление);
- `UpdateAction` — вид операции по исправлению ошибочной записи; программист устанавливает значение этого параметра в зависимости от требуемых действий:
  - `uaAbort` (отмена изменения записи без выдачи сообщения);
  - `uaApplied` (удаление записи из кэша);
  - `uaFail` (отмена изменения записи с выдачей сообщения);
  - `uaRetry` (повтор операции сохранения);
  - `uaSkip` (пропуск записи);
- `E` — объект-исключение, содержащий описание возникшей ошибки; в первую очередь используются свойства `ErrorCode` и `Message` этого объекта, которые определяют код ошибки и текстовое сообщение соответственно (обработка объекта-исключения была рассмотрена в *главе 7*).

Например, в процедуре:

```
procedure TForm1.Query1UpdateError(DataSet: TDataSet;
      e: EDatabaseError; UpdateKind: TUpdateKind;
      var UpdateAction: TUpdateAction);
begin
  UpdateAction := uaFail;
end;
```

при подтверждении кэшированных изменений набора данных Query1 отменяются изменения, сделанные во всех записях, которые приводят к ошибке.

Более сложно организовать исправление ошибочных записей. Доступ к значениям столбцов (полей) набора данных имеет особенность, связанную с тем, что существуют две версии записи: старая и неподтвержденная новая. Для доступа к старому и новому значениям столбца текущей записи предназначены соответственно свойства NewValue и OldValue.

Рассмотрим следующий пример:

```
procedure TForm1.Query1UpdateError(DataSet: TDataSet;
      e: EDatabaseError; UpdateKind: TUpdateKind;
      var UpdateAction: TUpdateAction);
begin
  Query1.FieldByName('Code').NewValue :=
      Query1.FieldByName('Code').OldValue;
  UpdateAction := uaRetry;
end;
```

При редактировании записей набора данных вносились изменения в столбец Code. При возникновении ошибки подтверждения изменений в столбце текущей записи восстанавливается его предыдущее значение. После этого операция сохранения изменений подтверждается.

Замечание

Если ошибочная ситуация в обработчике события OnUpdateError не устранена, то при повторных попытках сохранения записи происходит зацикливание программы.

В приведенном примере параметр DataSet обработчика не используется, а имя Query1 набора данных указано явно. Если обработчик является общим (разделяемым) для нескольких наборов данных, то вместо имени конкретного набора данных можно указать конструкцию вида

```
(DataSet as TDataSet)
```

или

```
TDataSet(DataSet)
```

Обе конструкции приводят тип параметра DataSet к типу набора данных TDataSet, первая — неявно, вторая — явно. Если тип набора данных известен более точно, например, TQuery, то его можно указать вместо типа TDataSet.

Компонент *UpdateSQL*

Компонент UpdateSQL предназначен для подтверждения кэшированных изменений в наборе данных Query. Главная задача этого компонента — обеспечить модификацию наборов данных, доступных только для чтения.

Набор данных Query связывается с компонентом UpdateSQL через свое свойство UpdateObject типа TDataSetUpdateObject, в качестве значения которого выбирается имя компонента UpdateSQL. Кроме того, для набора данных Query нужно установить в значение True свойства CachedUpdates и RequestLive.

Роль компонента UpdateSQL заключается в выполнении SQL-запросов, обновляющих необходимые записи связанного с ним набора данных Query. При этом данный компонент позволяет одновременно хранить и выполнять три разных запроса:

- UPDATE (редактирование записей);
- INSERT (вставка записей);
- DELETE (удаление записей).

Указанные запросы содержатся в свойствах ModifySQL, InsertSQL и DeleteSQL типа TStrings соответственно. Таким образом, компонент UpdateSQL частично совмещает функциональность сразу трех компонентов Query.

Ввести тексты соответствующих запросов компонента UpdateSQL можно вручную аналогично тому, как это выполняется для свойства SQL компонента Query. Однако на этапе разработки приложения более удобно использовать для этих целей Редактор свойств компонента UpdateSQL, окно которого показано на рис. 26.3. Редактор свойств вызывается командой **UpdateSQL Editor** контекстного меню компонента или двойным щелчком на компоненте UpdateSQL. В названии окна выводятся имена компонента UpdateSQL и соответствующего набора данных Query.

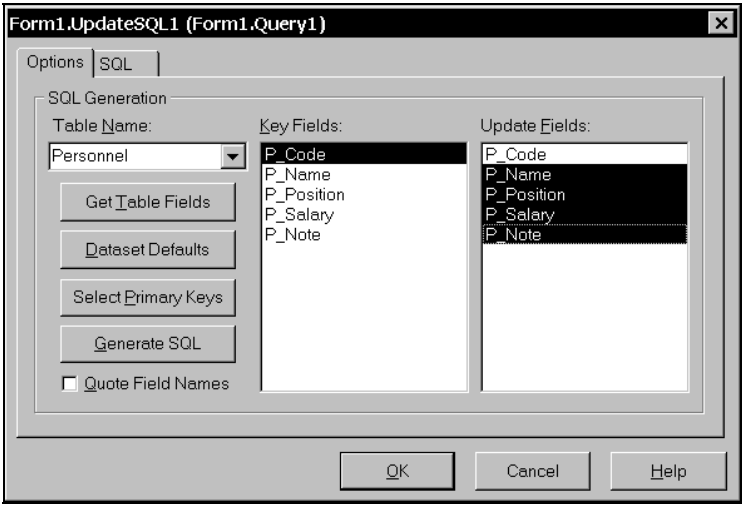


Рис. 26.3. Редактирование свойств компонента UpdateSQL

Замечание

Если для соответствующего набора данных Query не задан SQL-запрос, то элементы редактора свойств будут пустыми. Если SQL-запрос задан, то состояние набора данных (свойство Active) при выполнении этого запроса не имеет значения.

Редактор содержит две вкладки, первоначально выбрана вкладка параметров **Options**. В списке **Table Name** выбирается таблица, для записей которой включен механизм кэшированных изменений. Список содержит имена таблиц, указанных в запросе набора данных Query.

После выбора таблицы в списках ключевых (**Key Fields**) и обновляемых полей (**Update Fields**) отображается перечень ее столбцов. В первом списке нужно выделить столбцы, входящие в состав индекса и используемые при доступе к данным, а во втором — столбцы, значения которых будут обновляться.

Замечание

При работе с локальными таблицами Paradox нельзя выбирать поля автоинкрементного типа. На рис. 26.3 таким полем является ключевое поле `P_Code`. Нажатие кнопки **Select Primary Keys** выделяет ключевые поля таблицы.

Нажатие кнопки **Generate SQL** инициирует автоматическую генерацию SQL-запросов для выполнения обновления записей. Тексты запросов можно просмотреть и при необходимости изменить на вкладке **SQL** (рис. 26.4), переход на которую осуществляется после генерации запросов.

Группа переключателей **Statement Type** определяет тип запроса, текст которого содержится в поле редактирования **SQL Text**:

- **Modify** — запрос редактирования `UPDATE`;
- **Insert** — запрос вставки `INSERT`;
- **Delete** — запрос удаления `DELETE`.

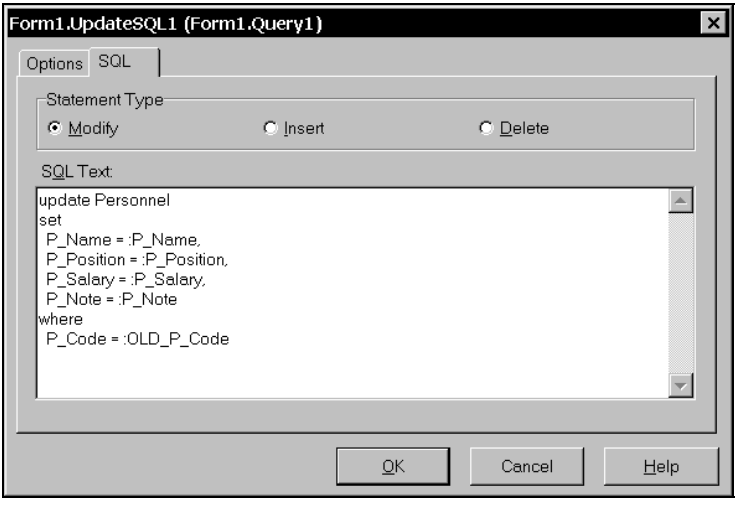


Рис. 26.4. SQL-запросы компонента `UpdateSQL`

Запросы компонента `UpdateSQL` можно выполнить либо через вызов методов компонента `Database`, либо через метод самого компонента `UpdateSQL`.

При подтверждении кэшированных изменений методом `ApplyUpdates` компонента `Database` для указанных наборов данных `Query` автоматически отрабатываются запросы соответствующего компонента `UpdateSQL`. Эти запросы и выполняют сохранение кэшированных изменений "своего" набора. Например:

```
Database1.ApplyUpdates([Query1]);
```

Напомним, что набор данных Query должен быть связан с компонентами Database и UpdateSQL через свои свойства DatabaseName и UpdateObject соответственно.

Подтверждение кэшированных изменений также можно выполнить последовательным вызовом методов SetParams и ExecSQL. Метод SetParams(UpdateKind: TUpdateKind) устанавливает для изменяемой записи значения параметров соответствующего SQL-запроса, а метод ExecSQL(UpdateKind: TUpdateKind) выполняет указанный запрос. Параметр UpdateKind, одинаковый для обеих процедур, определяет вид изменения записи:

- ukModify (редактирование);
- ukInsert (вставка);
- ukDelete (удаление).

Так как эти методы работают с текущей записью, они должны вызываться для каждой измененной записи набора данных. Поэтому вызов методов подтверждения выполняют в обработчике OnUpdateRecord типа TUpdateRecordEvent. Тип события описан следующим образом:

```
type TUpdateRecordEvent =
      procedure(DataSet: TDataSet; UpdateKind: TUpdateKind;
          var UpdateAction: TUpdateAction) of object;
```

Параметры этого события отличаются от параметров рассмотренного ранее события OnUpdateError отсутствием параметра E, содержащего объект-исключение.

Пример использования компонента UpdateSQL:

```
procedure TForm1.Query1UpdateRecord(DataSet: TDataSet;
            UpdateKind: TUpdateKind; VAR UpdateAction: TUpdateAction);
begin
  UpdateSQL1.SetParams(UpdateKind);
  UpdateSQL1.ExecSQL(UpdateKind);
  UpdateAction := uaApplied;
end;
```

Параметр UpdateAction устанавливается в значение uaApplied, что соответствует подтверждению изменений в записях.

Последовательный вызов методов SetParams и ExecSQL можно заменить вызовом метода Apply(UpdateKind: TUpdateKind), объединяющего их функциональность.

В данном разделе мы рассмотрели использование для одного набора данных Query одного компонента UpdateSQL, связь с которым устанавливается через свойство UpdateObject набора данных. Если с одним набором данных нужно связать несколько компонентов UpdateSQL, то следует обратиться к их свойствам DataSet, которые должны указывать на набор данных Query.

Механизм событий сервера

Механизм событий (сообщений) может использоваться сервером для информирования клиентских приложений о том, что произошло определенное событие, например, сбой в системе. При возникновении события всем активным приложениям посылается соответствующее сообщение.

Для этого требуется:

- создать на сервере триггер или хранимую процедуру;
- зарегистрировать сообщение в приложении.

В триггере или хранимой процедуре отправку сообщения выполняет инструкция `POST_EVENT` отправки сообщения, имеющая формат:

```
POST_EVENT "<имя события>";
```

Рассмотрим пример (листинг 26.1).

Листинг 26.1. Пример отправки сообщения в триггере

```
CREATE TRIGGER InsertStore FOR Store
ACTIVE
AFTER INSERT
      AS
      BEGIN
            POST_EVENT "evChangeStore";
      END

CREATE TRIGGER DeleteStore FOR Store
ACTIVE
AFTER DELETE
      AS
      BEGIN
            POST_EVENT "evChangeStore";
      END
CREATE TRIGGER UpdateStore FOR Store
ACTIVE
AFTER UPDATE
      AS
      BEGIN
            POST_EVENT "evChangeStore";
      END
```

При модификации (вставке, удалении или редактировании) записей таблицы приложениям посылается сообщение `evChangeStore`. Для этого созданы триггеры, автоматически реагирующие на соответствующие события.

Если для отправки сообщения используется не триггер, а хранимая процедура, которая, в отличие от триггера, не выполняется автоматически, то ее можно вызывать как из сервера, например из триггера или другой процедуры, так и из приложения. В последнем случае можно организовать обмен сообщениями через сервер, в том числе для задач, которые не работают с удаленной БД сервера. В этом случае сервер (сервер БД) используется приложениями как своеобразный диспетчер.

Для регистрации сообщений в приложении предназначен компонент `IBEvents`, свойство `Database` которого указывает на компонент `Database`, используемый для соединения с БД.

Замечание

В предыдущей версии Delphi в качестве основного средства регистрации сообщений в приложении применялся компонент `IBEventAlerter`, располагавшийся на изъятой теперь странице **Samples** Палитры компонентов.

Свойство `Events` типа `TStrings` определяет список событий, которые может обрабатывать приложение. При разработке приложения перечень событий редактируется в окне **EventAlerter Events** (рис. 26.5).

Рис. 26.5. Список событий компонента `IBEvents`

На этапе выполнения приложения списком сообщений можно управлять динамически, используя возможности класса `TSrtings`.

Например, команда

```
IBEvents1.Events.Add('evChangeStore');
```

добавляет событие `evChangeStore` к списку событий компонента `IBEventAlerter1`.

После определения списка интересующих приложение событий их следует зарегистрировать в сервере. Регистрация выполняется вызовом метода `RegisterEvents`, в свою очередь метод `UnRegisterEvents` отменяет регистрацию событий. Например:

```
IBEventAlerter1.RegisterEvents;
```

Для задания реакции на поступающие от сервера события используется обработчик события `OnEventAlert` типа `TEventAlert`, описанного как

```
TYPE TEventAlert = PROCEDURE (Sender: TObject; EventName: String;
         EventCount: Integer; VAR CancelAlerts: Boolean);
```

Кроме параметра `Sender`, указывающего объект-источник события, обработчику передаются также следующие параметры:

- `EventName` — имя полученного события, например, `evChangeStore`;
- `EventCount` — число событий, произошедших в сервере с момента передачи предыдущего события;

- `CancelAlerts` — режим приема событий: значение `True` (по умолчанию) означает, что сервер по-прежнему должен присылать сообщения о таком событии, значение `False` соответствует прекращению отправки таких событий.

Обработка поступившего события реализуется программистом в зависимости от характера события и назначения приложения.

Рассмотрим следующий пример (листинг 26.2).

Листинг 26.2. Пример обработки события

```
VAR CountChange: Integer = 0;
...
// Регистрация события
PROCEDURE TForm1.FormCreate(Sender: TObject);
BEGIN
      IBEvents1.Events.Clear;
      IBEvents1.Events.Add('evChangeStore');
      IBEvents1.RegisterEvents;
END;
// Обработка события
PROCEDURE TForm1.IBEvents1EventAlert(Sender: TObject;
      EventName: String; EventCount: Integer;
      VAR CancelAlerts: Boolean);
BEGIN
IF EventName = 'evChangeStore' THEN BEGIN
      CountChange:= CountChange + EventCount;
      IF CountChange >= 30 THEN BEGIN
              Query1.Close;
              Query1.Open;
              CountChange := 0;
              END;
       END;
END;
```

При создании формы `Form1` приложения в список заносится событие `evChangeStore` и выполняется регистрация этого события в сервере. Обработка события заключается в следующем. В глобальной переменной `CountChange` подсчитывается число изменений таблицы `Store`. В случае превышения этим числом критического значения `30` обновляется набор данных `Query1`, который обеспечивает отбор записей из названной таблицы.

Управление привилегиями

Привилегии представляют собой права доступа к БД. Управление привилегиями заключается в их установке и удалении. После создания объекта БД (например, таблицы) доступ к ней разрешен только создателю и системному администратору, имеющему имя SYSDBA. Для доступа к БД остальных пользователей им нужно назначить соответствующие привилегии. Сразу после появления нового пользователя, созданного например, с помощью программы InterBase Manager Server, этот пользователь имеет ми-

нимальные права доступа: ему разрешено только войти в БД (соединиться с ней), указав свое имя и пароль, однако ни один объект этой БД ему не доступен. Чтобы обеспечить возможность активной работы с БД, нужно определить (переопределить) привилегии.

Установку привилегий выполняет инструкция

```
GRANT
<Список описателей вида доступа>
ON [TABLE] {<Имя таблицы> | <Имя просмотра>}
       TO {<Пользователь> | <Список пользователей>} |
       EXECUTE ON PROCEDURE <Имя процедуры>
            TO {<Пользователь> | <Список пользователей>};

<Пользователь> =
      PROCEDURE <Имя процедуры>
      | TRIGGER <Имя триггера>
      | VIEW    <Имя просмотра>
      | [USER]  <Имя пользователя>
      | PUBLIC

<Список пользователей> =
       [USER]   <Имя пользователя1>
       ...
       [,[USER] <Имя пользователяN>]
       [WITH GRANT OPTION]
```

Привилегии позволяют разграничить доступ к таблицам и просмотрам со стороны пользователей. При этом под "пользователем" понимается любой объект, обращающийся к данным. Кроме собственно пользователя (приложения), такими объектами могут быть таблицы, просмотры, хранимые процедуры и триггеры. Если привилегия предоставляется одновременно нескольким пользователям, то их имена перечисляются через запятую. Описатель `PUBLIC` означает, что привилегия устанавливается для всех пользователей. Описатель `WITH GRANT OPTION` означает, что пользователь может устанавливать предоставленные ему привилегии другим пользователям.

В качестве описателей, определяющих вид доступа, указываются следующие:

- `ALL` (все права доступа);
- `SELECT` (только чтение);
- `DELETE` (удаление);
- `INSERT` (вставка);
- `UPDATE` (модификация).

После описателя `UPDATE` в круглых скобках можно указать список редактируемых столбцов таблицы или просмотра. Если требуется определить несколько прав, то соответствующие описатели перечисляются через запятую.

Для установки привилегий, кроме вида доступа, нужно указать еще имя таблицы или просмотра, а также имя пользователя. Остальная информация является необязательной.

Пример установки привилегий:

```
GRANT ALL ON Personnel TO Chief;
GRANT SELECT ON Personnel TO Manager;
GRANT SELECT, UPDATE (Name, BirthDay) ON Personnel TO TopManager;
```

Права доступа устанавливаются к таблице Personnel. Пользователь с именем Chief получает полные права доступа к таблице. Пользователь с именем Manager может только читать данные, а пользователь TopManager, кроме того, имеет право редактировать столбцы Name и BirthDay.

Еще один пример установки прав доступа:

```
GRANT ALL ON Store TO Chief, Manager, TopManager;
GRANT SELECT ON List TO PUBLIC;
```

Здесь полные права доступа к таблице Store получают три пользователя: Chief, Manager и TopManager. Читать записи из таблицы List разрешается всем пользователям.

Удаление привилегий заключается в отмене заданного ранее права доступа. Отмену привилегии выполняет инструкция REVOKE, формат которой аналогичен формату инструкции GRANT установки привилегии. Отличие заключается в том, что инструкция дополнительно имеет необязательный описатель GRANT OPTION FOR, который удаляет не саму привилегию, а право выдачи ее другим пользователям. Отметим, что удалить привилегию может только тот, кто ее установил.

Так, после выполнения команды

```
REVOKE ALL ON Personnel TO Chief;
```

пользователь с именем Chief лишается права доступа к таблице Personnel.

Манипулирование данными

Инструкции манипулирования данными предназначены для отбора и изменения данных. Отбор записей выполняется с помощью инструкции SELECT. Изменение данных заключается в редактировании, удалении и вставке записей, для чего используются инструкции UPDATE, INSERT и DELETE соответственно. Применительно к локальным БД указанные инструкции рассмотрены в *главе 20*, посвященной реляционному способу доступа с помощью механизма BDE. В этом разделе мы познакомимся с особенностями отбора записей в удаленных БД с помощью инструкции SELECT, которая имеет следующий формат:

```
SELECT [DISTINCT | ALL] {* | <Список выражений>}
       FROM <Список таблиц>
       [WHERE <Условия выбора>]
       [ORDER BY <Список столбцов для сортировки>]
       [GROUP BY  <Список столбцов для группирования>]
       [HAVING <Условия группирования>]
       [UNION <Вложенная инструкция SELECT>]
       [PLAN <План выполнения запроса>]
```

По сравнению с локальной версией дополнительными здесь являются операнд PLAN, задающий план выполнения запроса, и описатель ALL. Описатель ALL по своему действию противоположен описателю DISTINCT, обеспечивающему отбор уникальных записей (т. е. записей, значения столбцов которых не повторяются). Применение описателя ALL позволяет отобрать записи с повторяющимися значениями. Явно использовать этот описатель нет необходимости, т. к. он действует по умолчанию.

Кроме того, по сравнению с SQL-запросом для локальных БД условия отбора записей могут быть более сложными.

Формат условий следующий:

```
<Условие отбора>] =
      {<Выражение> <Операция сравнения> {<Выражение1> | (<Отбор1>)} |
      <Выражение> [NOT] <Мин. значение> AND <Макс. значение> |
      <Выражение> [NOT] LIKE < Выражение1> [ESCAPE < Выражение2>] |
      <Выражение> [NOT] IN
       (<Выражение1> [, <Выражение2>]...[, <ВыражениеN>] | (<ОтборM>)) |
      <Выражение> IS [NOT] NULL |
      <Выражение> [NOT] {<Операция сравнения> | ALL | SOME | ANY} (<ОтборM>) |
      EXISTS (<ОтборM>) |
      SINGULAR (<ОтборM>) |
      <Выражение> [NOT] CONTAINING <Выражение1> |
      <Выражение> [NOT] STARTING [WITH] <Выражение1>
```

Элементы, обозначенные как `Отбор`, представляют собой совокупность значений, отобранных с помощью инструкции `SELECT`, т. е. вложенную инструкцию `SELECT`. Заданное в этой инструкции условие отбора записей для элемента `Отбор1` возвращает одно значение или ни одного, а для элемента `ОтборM` — несколько значений, в частности, также ни одного.

Условие отбора может состоять из нескольких условий, связанных логическими операциями `NOT`, `OR` и `AND`.

Рассмотрим операнды, которые не встречались в локальной версии SQL.

Операнд `<Выражение> [NOT] CONTAINING <Выражение1>` позволяет отобрать записи на основании частичного совпадения значений. С его помощью выбираются записи, для которых значение `<Выражение>` включает в себя значение `<Выражение1>`. Обычно этот операнд применяется для строк. Например, с помощью инструкции

```
SELECT * FROM Store
       WHERE S_Name CONTAINING "то"
```

для таблицы `Store` склада выводятся все столбцы с данными о товарах, названия которых содержат символы `то`. Указанные символы могут входить в название товара с любой позиции, не обязательно с первой. Поэтому будут отобраны, например, `томаты` и `картофель`. Выражения заданы простейшими способами — указанием имени столбца и строкового значения. В общем случае выражения могут быть более сложными и состоять из нескольких операндов, знаков операций и круглых скобок.

Операнд `<Выражение> [NOT] STARTING [WITH] <Выражение1>` отличается от операнда `CONTAINING` тем, что требует вхождения, начиная с начала выражения.

Напомним, что отбор записей с частичным совпадением значений также можно выполнить, используя операнд `LIKE`, рассмотренный в *главе 20*.

Операнды `EXISTS` и `SINGULAR` позволяют выполнить проверку числа записей, возвращаемых указанным в них подзапросом. Операнд `EXISTS (<ОтборM>)` имеет значение `True` в случае, если при отборе записей возвращено непустое множество значений.

Например, инструкцией

```
SELECT S_Name, S_Number
FROM Store
  WHERE EXISTS
    (SELECT C_Move
       FROM Cards
       WHERE C_Date BETWEEN "1.10.2002" AND "31.10.2002")
```

из таблицы `Store` склада отбираются названия и количества товаров, для которых было движение (приход или расход) за октябрь 2002 года. Движение товара анализируется с помощью запроса к таблице `Cards`.

Операнд `SINGULAR (<ОтборМ>)` имеет значение `True` в случае, если при отборе записей возвращено только одно значение. Например, так

```
SELECT Name
  FROM Personnel
  WHERE SINGULAR
    (SELECT Position
       FROM Position
       WHERE Position.Code = Personnel.Code)
```

из таблицы `Personnel` отбираются фамилии сотрудников организации, которые занимают только одну должность.

Операнд `PLAN` позволяет задать план выполнения запроса:

```
<План выполнения запроса> =
       [{JOIN | [SORT] MERGE}]
       (<Таблица>) | <План выполнения запроса>,
      ...
       [(<Таблица>) | <План выполнения запроса>])

<Таблица> = {<Имя таблицы> | <Псевдоним таблицы>}
      {NATURAL | INDEX (<Список индексов>) | ORDER <Список индексов>}
```

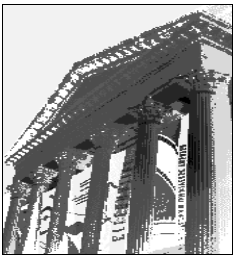
Задание плана выполнения запроса позволяет управлять методами доступа к данным и может увеличить скорость выполнения запроса. Для каждой таблицы, которая задается именем или псевдонимом, указанным после слова `FROM` инструкции `SELECT`, с помощью операндов можно определить:

- `NATURAL` — использование метода последовательного доступа к данным; если при поиске данных отсутствуют соответствующие индексы, то используется последовательный метод доступа;
- `INDEX` — использование для доступа к данным индексно-последовательного метода на основе указанных индексов;
- `ORDER` — сортировку таблицы по указанным индексам.

Описатель `JOIN` указывает, что план относится к связанным таблицам. Если для связанных таблиц существуют соответствующие индексы, то с их помощью будет реализован индексно-последовательный метод доступа. Если таких индексов нет, то для связанных таблиц следует указать описатель `MERGE` или `SORT MERGE`.

Инструкции `UPDATE`, `INSERT` и `DELETE` языка SQL сервера InterBase не имеют существенных отличий от своих аналогов, реализованных для локальной версии.

ГЛАВА 27



Технология InterBase Express

Общая характеристика

Технология InterBase Express ориентирована строго на работу с сервером InterBase версии не ниже 5.5. Отсюда следуют основные достоинства и недостатки этой технологии.

Преимущества технологии InterBase Express заключаются в том, все необходимые функции обеспечиваются путем прямого применения функций API сервера InterBase. В результате не нужно использовать BDE, повышается скорость работы компонентов доступа к данным.

Недостатком технологии InterBase Express является невозможность использовать серверы баз данных, отличные от сервера InterBase SQL Server.

Компоненты, предназначенные для работы по технологии InterBase Express, расположены на странице **InterBase** Палитры компонентов *(см. главу 14)*. Охарактеризуем кратко назначение основных из этих компонентов:

- `IBTable` — для получения данных из таблицы или представления базы данных. Полученный с помощью этого компонента набор данных является редактируемым. Является аналогом компонента `Table` для BDE, совместим с визуальными компонентами;
- `IBQuery` — для получения данных с помощью SQL-запроса. Является аналогом компонента `Query` для BDE, совместим с визуальными компонентами;
- `IBStoredProc` — для вызова хранимых процедур и получения набора данных на основе результатов выполнения процедуры. Соответствующий набор данных является нередактируемым. Совместим с визуальными компонентами;
- `IBDatabase` — для установления соединения с базой данных;
- `IBTransaction` — для управления транзакцией;
- `IBUpdateSQL` — для создания модифицируемых наборов данных, основанных на SQL-запросах. Является аналогом компонента `UpdateSQL` для BDE. Используется совместно с компонентом `IBQuery`;

- `IBDataSet` — для получения и редактирования данных. Совместим со всеми визуальными компонентами. Обеспечивает эффективный доступ к данным для просмотра и редактирования;
- `IBSQL` — для быстрого выполнения SQL-запроса с минимальными накладными расходами. Не имеет локального буфера данных, не совместим с визуальными компонентами;
- `IBDatabaseInfo` — для получения системной информации о свойствах базы данных, соединения и сервера;
- `IBSQLMonitor` — для перехвата и отслеживания SQL-запросов, которые выполняют приложения по технологии InterBase Express;
- `IBEvents` — для обработки событий сервера InterBase;
- `IBExtract` — для получения метаданных от сервера InterBase;
- `IBClientDataSet` — для получения данных и применения обновлений. Использует внутренние компоненты `TIBDataSet` и `TDataSetProvider`.

Компоненты `IBTable`, `IBQuery`, `IBStoredProc` и `IBUpdateSQL` во многом похожи на свои аналоги, рассмотренные при описании механизма BDE. Поэтому мы опишем свойства и методы, раскрывающие отличительные особенности их использования в технологии InterBase Express.

Установление соединения с сервером

За установление соединения с сервером БД отвечает компонент `IBDataBase`, являющийся аналогом компонента `DataBase` в технологии BDE и компонента `ADOConnection` в технологии ADO.

Имя базы данных, с которой устанавливается соединение, определяет свойство `DatabaseName` компонента `IBDataBase`. В случае локального набора данных InterBase это может быть имя файла базы данных с расширением gdb. Нажатие кнопки с многоточием в строке этого свойства в окне Инспектора объектов приводит к вызову стандартного диалога открытия файлов, с помощью которого можно выбрать нужный файл. В случае установления соединения с базой данных InterBase на удаленном сервере по протоколу TCP/IP значение свойства задается в виде:

```
<имя сервера>:<имя файла>
```

Параметры соединения с сервером можно установить также в диалоговом окне **Database Component Editor** (рис. 27.1), открываемом двойным щелчком на рассматриваемом компоненте, либо выбором пункта **Database Editor** его контекстного меню. В поле **Database** требуется задать спецификацию файла БД, а в поля **User Name** и **Password** — ввести имя пользователя и пароль.

Заданные таким образом параметры соединения автоматически отображаются в поле **Settings** рассматриваемого диалогового окна, они же являются значением свойства `Params` типа `TStrings` компонента `IBDataBase`.

С помощью флажка **Login Prompt** в окне **Database Component Editor** (см. рис. 27.1) задается необходимость появления окна запроса для указания имени пользователя и па-

роля при запуске приложения. При отказе от необходимости указания имени пользователя и пароля следует сбросить этот флажок.

Для проверки правильности соединения нужно нажать кнопку **Test**. По результатам тестирования выдается соответствующее сообщение.

Открыть соединение с базой данных можно в Инспекторе объектов при разработке приложения или при выполнении приложения, задав свойству `Connected` типа `Boolean` значение `True`. При выполнении приложения для открытия и закрытия соединения с базой данных можно воспользоваться методами `Open` и `Close` соответственно.

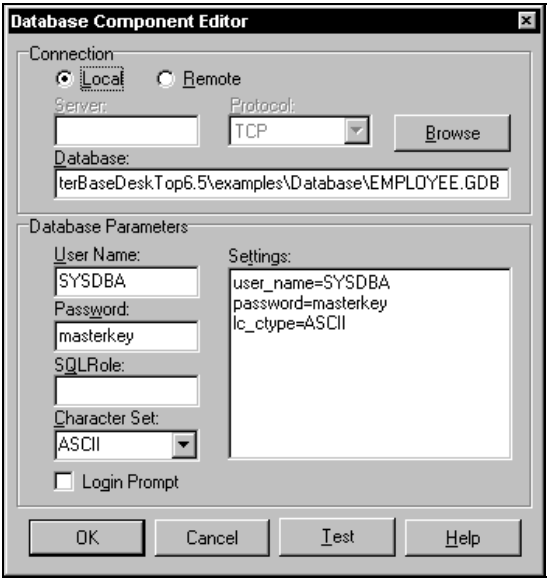


Рис. 27.1. Диалоговое окно **Database Component Editor**

Состояние соединения с сервером БД в ходе выполнения приложения можно проверить с помощью метода `CheckActive` или `CheckInactive` (противоположный вариант предыдущего метода).

Для получения списка имен таблиц в БД служит метод `GetTableNames(List: TStrings; SystemTables: Boolean = False)`. При задании параметру `SystemTables` значения `False` в этот список не попадут имена системных таблиц.

Для получения списка имен полей требуемой таблицы БД предназначен метод `GetFieldNames(const TableName: string; List: TStrings)`. Имя таблицы определяет параметр `TableName`.

Управление транзакциями

Для управления транзакциями при работе с сервером InterBase служит компонент `IBTransaction`. В разрабатываемом приложении БД может быть несколько компонентов `IBDataBase`. Для компонента `IBTransaction` свойства `Databases[Index: Integer]` типа

TIBDatabase и DatabaseCount типа Integer содержат соответственно список и общее число связанных с ним компонентов IBDataBase. Свойство DefaultDatabase типа TIBDatabase определяет, какой из компонентов IBDataBase используется по умолчанию при выполнении транзакции.

В свою очередь, в приложении может быть несколько компонентов управления транзакциями. Для компонента IBDataBase транзакцию по умолчанию можно задать через его свойство DefaultTransaction типа TIBTransaction. Его свойства Transactions[Index: Integer] типа TIBTransaction и TransactionCount типа Integer содержат список и общее число связанных с ним транзакций соответственно.

При выполнении приложения для транзакции можно установить и разорвать связь с соединением с помощью методов AddDatabase и RemoveDatabase соответственно.

Для запуска, фиксации и отката транзакции соответственно служат методы: StartTransaction, Commit и Rollback. Начиная с версии 6 сервер InterBase поддерживает два новых метода: CommitRetaining и RollbackRetaining. Они оставляют транзакцию открытой после фиксации и после отката текущей транзакции соответственно.

Перед запуском транзакции целесообразно определить значение свойства InTransaction типа Boolean. Значение True соответствует незаконченной предыдущей транзакции, и запуск новой транзакции без завершения предыдущей путем фиксации или отката приведет к исключению.

Допустимое время ожидания отклика на запущенную транзакцию задает свойство IdleTimer типа Integer. Если за указанный промежуток времени транзакция не будет завершена, то будет выполнено действие по умолчанию. Его определяет свойство DefaultAction типа TTransactionAction, который описан так:

```
type  TTransactionAction = (taRollback, taCommit, taRollbackRetaining,
                            taCommitRetaining);
```

Здесь:

- taRollback — откат транзакции;
- taCommit — фиксация транзакции;
- taRollbackRetainong — откат транзакции с сохранением ее контекста (начиная с версии 6 InterBase);
- taCommitRetaining — фиксация транзакции с сохранением ее контекста (начиная с версии 6 InterBase).

При управлении транзакциями целесообразно установить один из четырех возможных вариантов так называемого *уровня изоляции транзакции*. Удобно сделать это с помощью редактора Transaction Editor (рис. 27.2). Вызов его осуществляется посредством одноименной команды контекстного меню компонента IBTransaction.

Уровень изоляции транзакции определяет, какие изменения, сделанные в других транзакциях, может видеть настраиваемая транзакция. Из числа возможных вариантов в окне редактора Transaction Editor обычно рекомендуется установка переключателя **Read Committed**. При этом запросы в одной транзакции могут просматривать измене-

ния, внесенные и подтвержденные в контексте конкурирующих транзакций. Этот вариант уровня изоляции задается константой `read_committed` (возможность читать подтвержденные записи других транзакций). Такой уровень изоляции транзакций чаще всего используется для получения последнего состояния базы данных. Имеются две разновидности этого уровня изоляции: `rec_version` и `no_rec_version`. Вариант с параметром `rec_version` используется по умолчанию. Это означает, что при считывании записи считывается ее последняя подтвержденная версия.

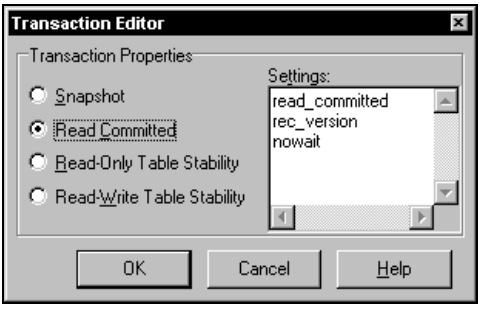


Рис. 27.2. Окно **Transaction Editor**

Вариант **Snapshot** задается параметром `concurrency` (транзакция производит снимок маски транзакции в момент запуска). Изменения, производимые конкурирующими транзакциями, ей не видны. Ей доступны только свои изменения. Обычно такой вариант применяется для продолжительных запросов или для блокирования записей, чтобы предотвратить их изменение другими транзакциями.

Варианты **Read-Only Table Stability** и **Read-Write Table Stability** задаются параметром `consistency` (аналогичен параметру `concurrency`, дополнительно блокирует запись в таблицу). Указанные варианты различаются режимами доступа, которые соответственно задаются параметрами `read` (разрешает только операции чтения) и `write` (разрешает операции записи). Использование такого уровня изоляции поддерживает последовательные (сериализуемые) обновления таблицы. Такой уровень обычно применяется для коротких обновляющих транзакций.

Компоненты доступа к данным

Стандартные компоненты (`IBTable`, `IBQuery`, `IBUpdateSQL` и `IBStoredProc`) доступа к данным по технологии InterBase Express наследуют механизм от родительского класса `TIBCustomDataSet`. Поэтому прежде всего представляет интерес рассмотрение основных его свойств и методов.

Стандартные компоненты доступа к данным подключаются к БД через компонент соединения `IBDataBase` с помощью своего свойства `Database` типа `TIBDatabase`.

Тип записей набора данных, для которых применима операция кэширования, определяет свойство `UpdateRecordTypes` типа `TIBUpdateRecordTypes`, который описан так:

```
type  TIBUpdateRecordTypes = set of (cusModified, cusInserted, cusDeleted,
                                     cusUnmodified, cusUninserted);
```

Здесь:

- `cusModified` — измененные записи;
- `cusInserted` — добавленные записи;
- `cusDeleted` — удаленные записи;
- `cusUnmodified` — неизмененные записи;
- `cusUninserted` — недобавленные записи.

Свойство `ForcedRefresh` типа `Boolean` определяет, будет ли выполняться обновление набора данных при каждом сохранении внесенных изменений. Для ускорения работы приложения рекомендуется задать этому свойству значение `False` (действует по умолчанию). Чтобы обновление набора данных происходило как можно чаще, следует этому свойству задать значение `True`.

Генераторы для автоинкрементных полей

У ряда компонентов доступа к данным, реализующим запросы (например, `IBQuery` и `IBDataSet`), свойство `GeneratorField` типа `TIBGeneratorField` позволяет присваивать значения первичным ключам набора данных с помощью генератора, назначаемого с использованием редактора названного свойства.

С помощью этого свойства можно обеспечить автоматическое задание значения одного из автоинкрементных полей, указав при этом, какое поле должно получать значение и как вычисляться (шаг изменения и событие: вставка записи, сохранение записи, по команде сервера). Для задания рассматриваемого свойства используется редактор (рис. 27.3), вызываемый двойным щелчком мыши в строке свойства в окне Инспектора объектов.

В окне редактора следует выбрать название генератора, имя автоинкрементного поля в таблице, для которого будут генерироваться значения, и шаг увеличения счетчика (обычно 1). Кроме того, нужно выбрать вариант события, при наступлении которого будет происходить генерация значения поля: **On New Record** (при вставке новой записи), **On Post** (при сохранении записи) и **On Server** (по команде сервера с помощью триггера).

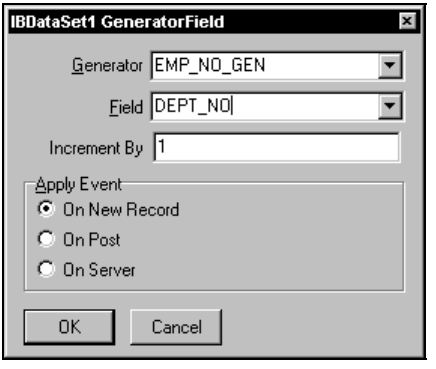


Рис. 27.3. Окно редактора свойства `GeneratorField`

Доступ к таблицам

Для доступа к таблицам по технологии InterBase Express служит компонент `IBTable`. Рассмотрим кратко отличительные свойства этого компонента.

Размер буфера по числу записей определяет свойство `BufferChunks` типа `Integer`.

Тип отношений, доступных в дополнение к пользовательским таблицам при выборе в раскрывающемся списке с помощью свойства `TableName`, определяет свойство `TableTypes` типа `TIBTableTypes`. Оно может принимать следующие значения:

- `ttSystem` — доступны системные таблицы и представления;
- `ttView` — доступны пользовательские представления.

Существует ли в БД таблица, указанная с помощью свойства `TableName`, определяет свойство `Exists` типа `Boolean`. Это свойство рекомендуется проверять перед открытием таблицы при выполнении приложения.

При использовании компонента `IBTable` указывается только имя таблицы с помощью свойства `TableName`. При этом автоматически формируется запрос на выборку всей таблицы со всеми полями. Если в таблице имеется очень много записей, то после выполнения такого запроса на сервер может лечь чрезмерно большая нагрузка. К примеру, при выполнении операций поиска нужной записи. Поэтому в крупных программных приложениях БД использование компонента `IBTable` не рекомендуется.

Выполнение запросов

Для выполнения SQL-запросов по технологии InterBase Express служит компонент `IBQuery`. Этот компонент является аналогом компонента `Query` в технологии BDE. Рассмотрим кратко отличительные свойства компонента `IBQuery`.

Установить значение свойства `SQL` типа `TStrings`, определяющего текст исполняемого SQL-запроса, можно с помощью редактора CommandText Editor, работа с которым описана в *главе 9*.

Параметры исполняемого SQL-запроса хранятся в свойстве `Params` типа `TParams`, а общее число параметров запроса возвращает свойство `ParamCount` типа `Word`.

Специалисты рекомендуют использовать компонент `Query` в основном при переносе имеющихся приложений под BDE на платформу технологии InterBase Express. При разработке новых приложений взамен него предлагается использовать компонент `IBDataSet`.

Получение и редактирование данных

В рассматриваемой технологии для выполнения SQL-запросов по выборке, вставке, удалению и обновлению записей в таблицах и представления полученных наборов данных используется компонент `IBDataSet`. Этот компонент реализует практически все возможности описанных выше компонентов доступа к данным по рассматриваемой технологии, именно его и целесообразно использовать при разработке приложений БД.

На этапе разработки и при выполнении приложения для компонента IBDataSet доступны следующие свойства типа TStrings, задающие различные типы запросов:

- InsertSQL — на добавление записей в набор данных;
- DeleteSQL — на удаление записей из набора данных;
- ModifySQL — на изменение записей в наборе данных;
- SelectSQL — запрос на выборку всех записей набора данных;
- RefreshSQL — на обновление текущей записи набора данных.

Тип запроса к набору данных определяется с помощью свойства StatementType типа TIBSQLTypes, который описан так:

```
type  TIBSQLTypes = set of (SQLUnknown, SQLSelect, SQLInsert, SQLUpdate,
                            SQLDelete, SQLDDL, SQLGetSegment, SQLPutSegment,
                            SQLExecProcedure, SQLStartTransaction, SQLCommit,
                            SQLRollback, SQLSelectForUpdate, SQLSetGenerator);
```

Здесь:

- SQLCommit — фиксация текущей (активной) транзакции;
- SQLDDL — выполнение команды DDL;
- SQLDelete — удаление записей в таблице или курсоре;
- SQLExecProcedure — вызов хранимой процедуры;
- SQLGetSegment — чтение сегмента BLOB;
- SQLInsert — добавление записей в таблицу;
- SQLPutSegment — запись сегмента BLOB;
- SQLRollback — восстановление состояния БД, предшествовавшего началу текущей транзакции;
- SQLSetForUpdate — хранимая процедура, устанавливается для обновления;
- SQLSetGenerator — установка нового значения для существующего генератора;
- SQLSelect — возвращение данных из одной или нескольких таблиц (SQL-запрос);
- SQLStartTransaction — начало новой транзакции;
- SQLUnknown — неизвестный тип SQL;
- SQLUpdate — изменение записи в таблице, представлении или курсоре.

В целом компонент IBDataSet предназначен для выполнения модифицирующих запросов с целью изменения данных, полученных с помощью запроса на выборку, задаваемого в свойстве SelectSQL.

Рассмотрим технику выполнения модифицирующих запросов с использованием свойств ModifySQL, InsertSQL, DeleteSQL и RefreshSQL компонента IBDataSet. Предположим, что с помощью запроса на выборку, заданного в свойстве SelectSQL, мы получили набор данных, содержащий записи с нужными полями заданной таблицы. Пусть нам требуется отредактировать содержимое одного из нескольких полей некоторой записи. При этом после внесения изменений, например, в компоненте DBGrid, эти

изменения будут внесены в локальный буфер, а в базе данных на сервере ничего не изменится. Для внесения соответствующих изменений в базе данных на сервере нужно выполнить оператор UPDATE языка SQL. Причем этот запрос должен быть указан в качестве значения свойства ModifySQL рассматриваемого компонента.

К примеру, если редактируемый набор данных для таблицы с именем 'COUNTRY' содержит поля с именами 'COUNTRY' и 'CURRENCY', то соответствующий запрос на изменение записи в свойстве ModifySQL может содержать следующий текст:

```
UPDATE COUNTRY
SET COUNTRY = :COUNTRY, CURRENCY = :CURRENCY
WHERE COUNTRY = :OLD_COUNTRY
```

В приведенном запросе вместо значений указаны параметры, названия которых совпадают с названиями полей редактируемой таблицы. Это означает, что при изменении содержимого полей записи рассматриваемый компонент автоматически установит значения всех параметров через текущие значения полей или некоторым другим образом. Префикс OLD_ в названии параметра означает, что в параметр будет подставлено старое содержимое поля до его изменения пользователем.

Разработчик должен указать, каким именно образом происходит определение параметров. К примеру, возможный вариант вызова оператора UPDATE с заданием параметров в обработчике события нажатия кнопки может иметь следующий вид:

```
procedure TForm1.Button1Click(Sender: TObject);
begin     {изменение записи}
with  IBDataSet1 do
  begin
    Edit;
    FieldByName('COUNTRY').AsString := Edit1.Text;
    FieldByName('CURRENCY').AsString := trim(FieldByName('CURRENCY').AsString);
    Post;
  end;
end;
```

Как видно из приведенного примера, вызов оператора UPDATE осуществляется с помощью метода Edit класса TIBDataSet. А именно, путем вызова метода Edit выполняется подготовка буфера текущей записи для редактирования, с помощью метода FieldByName задаются значения параметров оператора UPDATE и тем самым изменяются значения полей. С помощью метода Post осуществляется сохранение внесенных изменений в базе данных на сервере.

Для вставки и удаления записей в используемом нами классе TIBDataSet1 служат методы Insert и Delete соответственно. Для запросов, указываемых в свойствах InsertSQL и DeleteSQL, задается аналогичная последовательность действий.

Рассмотренные нами запросы для компонента IBDataSet называют *динамическими* или *"живыми"* (live query).

Важным достоинством рассматриваемого компонента является то, что после выполнения любого действия по редактированию набора данных он выполняет запрос, указанный в свойстве RefreshSQL. Этот запрос выполняет возвращение из базы данных

одной — текущей записи и предназначен для обновления полей этой записи после внесенных изменений. Для нашего примера в свойстве `RefreshSQL` должен быть следующий код:

```
SELECT COUNTRY, CURRENCY
  FROM COUNTRY
  WHERE COUNTRY = :COUNTRY
```

Пояснить достоинство такого механизма можно следующим образом. Если в базе данных используются триггеры, модифицирующие значения полей базы данных, то внесенные в результате выполнения триггеров изменения пользователю не видны. Это означает, что для просмотра возможных изменений нужно выбрать нашу запись из базы данных с помощью оператора `SELECT` или полностью повторить выборку всех записей с помощью запроса из свойства `SelectSQL`. Такой вариант работы с измененными записями реализован при работе с помощью механизма BDE. Использование свойства `RefreshSQL` в рассматриваемой технологии позволяет избежать необходимости повторной выборки записей всего набора данных.

Для автоматизации построения модифицирующих запросов, т. е. задания свойств `ModifySQL`, `InsertSQL`, `DeleteSQL` и `RefreshSQL` компонента `IBDataSet`, служит редактор. Вызов редактора выполняется с помощью команды **Dataset Editor** контекстного меню компонента `IBDataSet`. Причем делается это после того, как задано его свойство `SelectSQL`.

На вкладке **Options** окна редактора модифицирующих запросов (рис. 27.4) выполняется выбор таблицы из списка **Table Name**. Далее с помощью кнопки **Get Table Fields** осуществляется формирование списков **Key Fields** и **Update Fields**. В списке **Key Fields** нужно выделить ключевые поля, на основе которых будет задаваться условие `WHERE` в создаваемых запросах. Если в выбранной таблице ключевые поля существуют, то для их выделения нужно нажать кнопку **Select Primary Keys**.

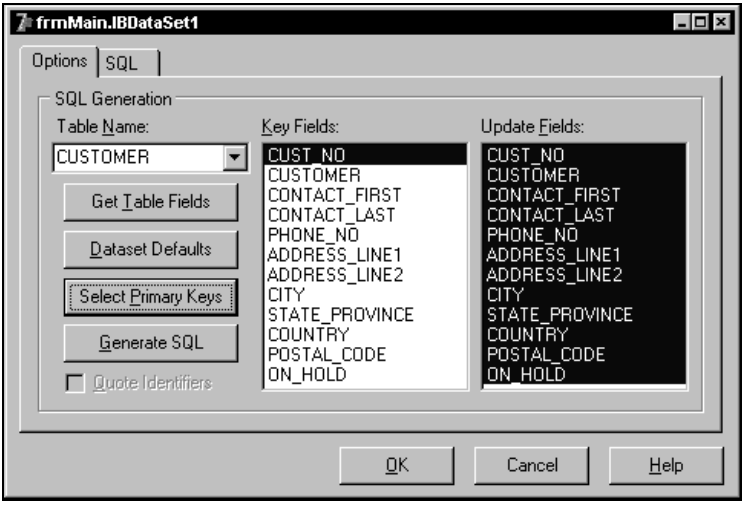


Рис. 27.4. Окно редактора модифицирующих запросов

Выделение полей, для которых допускается редактирование, выполняется с помощью списка **Update Fields**. В состав этого списка нельзя включать вычисляемые поля, поскольку их содержимое редактировать не допускается.

Нажатие кнопки **Generate SQL** вызывает генерацию запросов. На вкладке **SQL** окна редактора отображаются тексты автоматически сгенерированных модифицирующих запросов, т. е. значения свойств `ModifySQL`, `InsertSQL`, `DeleteSQL` и `RefreshSQL` компонента `IBDataSet`.

В компоненте `IBDataSet` реализованы обработчики трех событий:

```
property DatabaseDisconnected: TNotifyEvent;
property DatabaseDisconnecting: TNotifyEvent;
property DatabaseFree: TNotifyEvent;
```

Соответствующие события возникают после отключения БД, во время отключения БД и после высвобождения памяти компонентом соединения соответственно.

Компонент *IBSQL*

Компонент `IBSQL` обеспечивает объект для выполнения SQL-запроса с минимальными накладными расходами. Так, для каждого запроса к набору данных, заданного в компоненте `TIBDataSet`, формируется объект `TIBSQL`. Рассматриваемый компонент получает набор данных в виде однонаправленного курсора и не обеспечивает навигацию по набору данных. Компонент `IBSQL` не имеет локального буфера для набора данных и несовместим с визуальными компонентами.

Рассмотрим основные отличия компонента `IBSQL` от его аналогов для технологий BDE, dbExpress и ADO Express.

Список имен параметров запроса создает свойство `GenerateParamNames` типа `Boolean`, план запроса содержит свойство `Plan` типа `String`.

Число возвращенных после выполнения запроса записей содержит свойство `RecordCount` типа `Integer`.

Число записей, обработанных SQL-операторами `INSERT`, `DELETE` или `UPDATE`, содержит свойство `RowsAffected` типа `Integer`.

При работе приложения для баз данных зачастую полезно знать состояние набора данных. С этой целью при открытии запроса могут быть использованы следующие методы вызова исключений:

- `CheckClosed;` — если набор данных не закрыт;
- `CheckOpen;` — если набор данных не открыт;
- `CheckValidStatement;` — если запрос синтаксически неправилен.

Выскажем короткое соображение по тому, в каких случаях предпочтительно использовать компонент `IBSQL`. Для примера можно отметить вариант приложения, в котором требуется выполнять массовую вставку записей в базу данных и при этом нет необходимости отображать вставляемые записи в пользовательских компонентах. В этом случае целесообразно использовать компонент `IBSQL`, поскольку он не буферизует записи в отличие от `IBDataSet`. Тем самым мы достигаем более быстрой работы приложения.

Пример приложения

Рассмотрим пример приложения (рис. 27.5), построенного с использованием компонентов InterBase Express: `IBDataSet`, `IBDataBase`, `IBTransaction`, `DataSource` и нескольких вспомогательных компонентов `DBEdit`, `DBGrid` и `Button`.

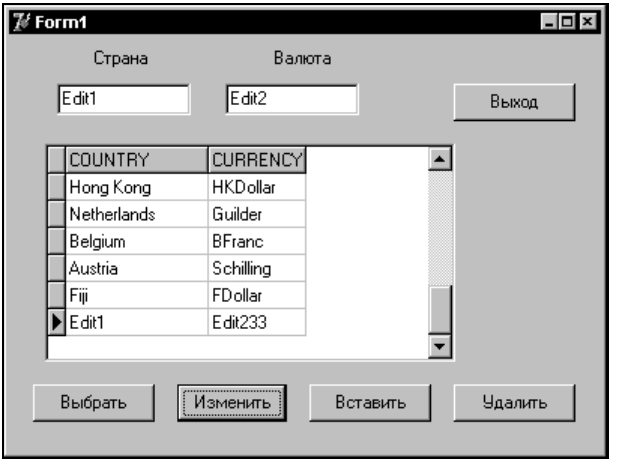


Рис. 27.5. Вид формы при выполнении приложения

В примере демонстрируется возможность выборки, изменения, вставки и удаления записей БД, отображаемой с помощью компонента `DBGrid`.

Код модуля формы `Unit1DataSet` для решения поставленной задачи приведен в листинге 27.1.

Листинг 27.1. Пример приложения с компонентами InterBase Express

```
unit Unit1DataSet;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, Grids, DBGrids, IBCustomDataSet, IBDatabase, StdCtrls;
type
  TForm1 = class(TForm)
    IBDatabase1: TIBDatabase;
    IBTransaction1: TIBTransaction;
    IBDataSet1: TIBDataSet;
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    Edit1: TEdit;
    Label1: TLabel;
    Edit2: TEdit;
    Label2: TLabel;
```

```
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    procedure Button2Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button2Click(Sender: TObject);
begin     {вставка записи}
  with  IBDataSet1 do
    begin
      Insert;
      FieldByName('COUNTRY').AsString := Edit1.Text;
      FieldByName('CURRENCY').AsString := Edit2.Text;
      Post;
    end;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin     {выборка записей}
  IBDataSet1.Active := False;
  IBDataSet1.Active := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin     {изменение записи}
with  IBDataSet1 do
  begin
    Edit;
    FieldByName('COUNTRY').AsString := Edit1.Text;
    FieldByName('CURRENCY').AsString := trim(FieldByName('CURRENCY').AsString);
    Post;
  end;
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin     {удаление записи}
  IBDataSet1.Delete;
end;

procedure TForm1.Button5Click(Sender: TObject);
begin     {выход}
  IBDataSet1.Active := False;
  Form1.Close;
end;
end.
```

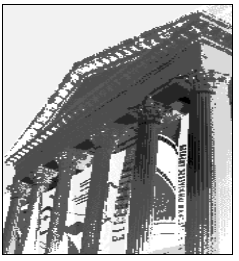
В обработчике события нажатия кнопки `Button4` (с заголовком **Выбрать**) компонент `IBDataSet1` устанавливается активным, тем самым выполняется запрос на выборку всех записей (свойство `SelectSQL`).

В обработчике события нажатия кнопки `Button1` (с заголовком **Изменить**) новое значение поля с именем `'COUNTRY'` берется из свойства `Edit1.Text`, в то время как новое значение поля с именем `'CURRENCY'` определяется как результат редактирования этого же поля.

В обработчике события нажатия кнопки `Button2` (с заголовком **Вставить**) с помощью метода `Insert` выполняется запрос из свойства `InsertSQL` на вставку записи в набор данных, здесь же задаются значения параметров запроса. Имена полей для вставляемой записи берутся из свойств `Edit1.Text` и `Edit2.Text` редакторов.

В обработчике события нажатия кнопки `Button3` (с заголовком **Удалить**) с помощью метода `Delete` выполняется запрос из свойства `DeleteSQL` на удаление текущей записи из набора данных.

ГЛАВА 28



Инструменты для работы с удаленными базами данных

Ранее были рассмотрены программные инструменты для работы с БД. Большинство этих программ можно использовать и для работы с удаленными БД. Так, создать псевдоним для InterBase позволяют программы Administrator BDE и Database Desktop, а редактировать и просматривать записи можно с помощью программы SQL Explorer. В дополнение к ним в данной главе мы познакомимся с программами, предназначенными специально для удаленных БД.

Программа IBConsole

Программа IBConsole предназначена для управления сервером InterBase и является его *консолью*. Консоль устанавливается совместно с сервером InterBase и находится в его каталоге BIN, ее главный файл называется IBConsole.exe. Программу также можно запустить через меню **Пуск** Windows, выбрав команду **Программы | InterBase | IBConsole**.

Для запуска и остановки сервера служит программа InterBase Server Manager, функциональность которой, начиная с версии Delphi 6, значительно уменьшилась.

Программа IBConsole обеспечивает:

- управление локальными и удаленными серверами;
- управление БД;
- интерактивное выполнение SQL-запросов.

Основную часть окна IBConsole занимают две панели (рис. 28.1). В левой панели в виде дерева представлены зарегистрированные серверы и их БД, а также элементы структуры, например, таблицы или пользователи. Отметим, что в качестве имени сервера или БД отображается его псевдоним (alias), задаваемый при регистрации сервера или БД (этот псевдоним никак не связан с псевдонимом BDE). В правой панели выводится информация об объекте, выбранном в левой панели.

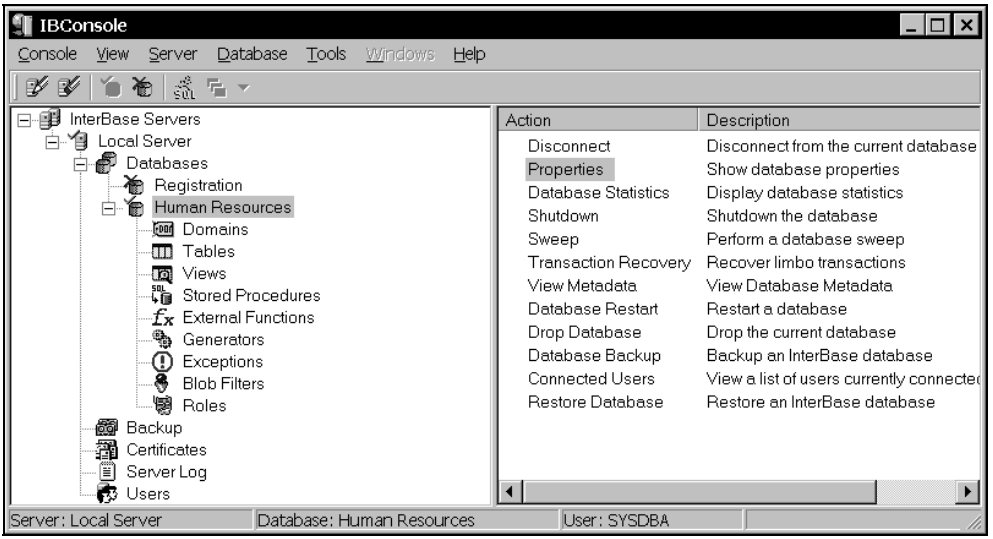


Рис. 28.1. Главное окно программы IBConsole

Управление сервером

Управление сервером заключается в:

- регистрации сервера;
- подключении сервера;
- просмотре протокола работы;
- управлении сертификатами;
- определении пользователей.

Для управления сервером используются команды меню **Server** главного окна программы IBConsole, а также команды контекстного меню сервера и его элементов.

После запуска консоли в левой панели отображается список зарегистрированных серверов, которые первоначально отключены, о чем свидетельствует красный крестик в значке сервера.

Подключение к серверу

Подключение к серверу, выбранному в левой панели, выполняется командой **Login**. При этом появляется окно **Server Login** (рис. 28.2), в котором необходимо указать имя пользователя (**User Name**) и его пароль (**Password**). После указания имени `SYSDBA` системного администратора, его пароля `masterkey` и нажатия кнопки **Login** осуществляется подключение к серверу, имя (alias) которого отображается в надписи **Server**, а к значку сервера добавляется зеленая галочка.

Отключение от выбранного сервера выполняется командой **Logout**. При этом выдается запрос на продолжение операции, и в случае подтверждения сервер отключается.

После подключения к серверу можно выполнить *проверку подключения* к одной из его БД. Командой **Diagnose Connection** открывается окно проверки соединения **Communication Diagnostics** (рис. 28.3), в котором указывается информация о сервере.

В качестве БД задается ее файл, который можно выбрать в окне **Open**, открываемом нажатием кнопки с тремя точками.

Кнопка **Test** инициирует проверку, результаты которой выводятся в поле **Results**.

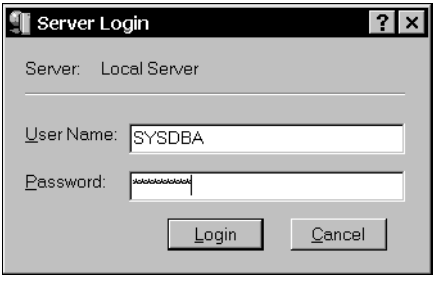


Рис. 28.2. Подключение к серверу

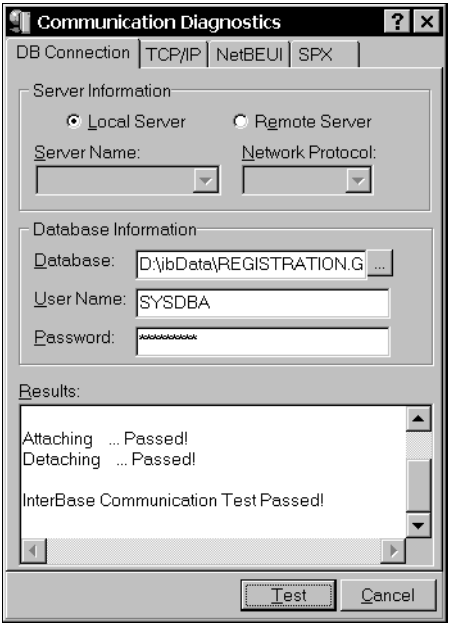


Рис. 28.3. Проверка подключения сервера к БД

На страницах **TCP/IP**, **NetBEUI**, **SPX** можно выполнить настройку соответствующего сетевого протокола.

Регистрация сервера

Для *регистрации* в консоли нового сервера необходимо выполнить команду **Register**, после чего открывается окно регистрации и соединения с сервером **Register Server and Connect** (рис. 28.4).

При регистрации локального сервера устанавливается переключатель **Local Server**. При необходимости в поле описания (**Description**) можно ввести краткую информацию, поясняющую назначение и особенности сервера. После нажатия кнопки **OK** локальный сервер регистрируется, а его имя добавляется к списку в левой панели.

В случае регистрации удаленного сервера устанавливается переключатель **Remote Server** и заполняются поля, которые заблокированы при подключении локального сервера: в поле **Server Name** указывается сетевое имя сервера, в списке **Network Protocol** выбирается протокол связи, а в поле **Alias Name** задается имя (псевдоним), под которым сервер регистрируется в консоли.

Одновременно с регистрацией можно выполнить подключение сервера, для чего должны быть заполнены поля **User Name** и **Password** группы **Login Information**.

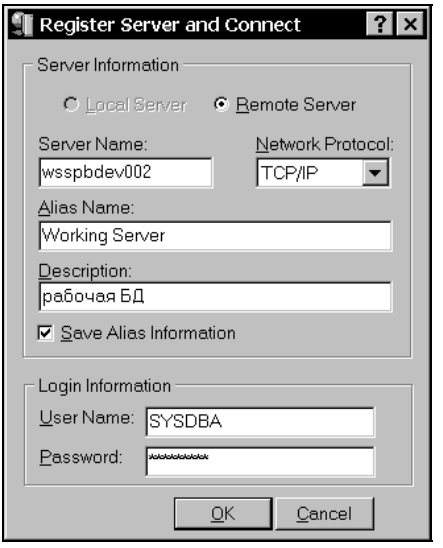


Рис. 28.4. Регистрация сервера

Отмена регистрации выбранного сервера выполняется командой **Un-Register**. При этом выдается запрос на продолжение операции, и в случае подтверждения сервер исключается из консоли. Перед отменой регистрации сервера его необходимо отключить.

Просмотр протокола работы сервера

В процессе работы сервера ведется *протокол* (log file), просмотреть который можно, вызвав команду **View Logfile**. При этом появляется окно протокола **Server Log** (рис. 28.5), в котором выводится краткий отчет о работе сервера.

Протокол можно распечатать или сохранить в текстовом файле.

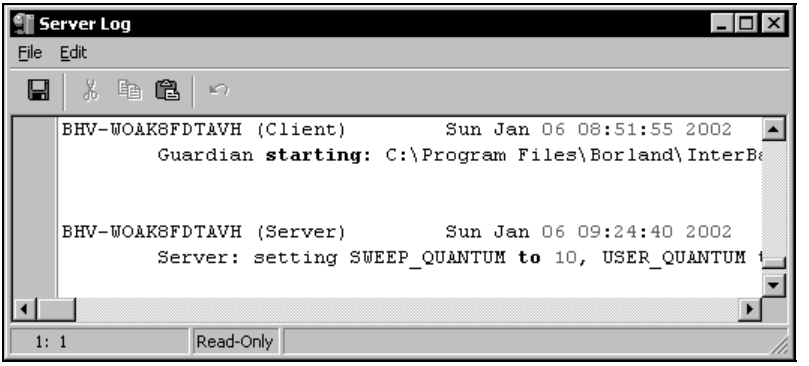


Рис. 28.5. Просмотр протокола работы сервера

Операции с сертификатами

Список сертификатов (certificate — удостоверение), действительных для сервера, можно просмотреть в правой панели, выбрав в левой панели элемент **Certificates**. Сертифи-

кат можно *добавить*, открыв командой **Add Certificate** одноименное окно **Add Certificate** (рис. 28.6) и указав в нем код (**ID**) и ключ (**Key**) сертификата. После нажатия кнопки **OK** выполняется проверка сертификата, и, если ошибки не найдены, новый сертификат добавляется к серверу.

Выбранный в списке сертификат *удаляется* командой **Remove Certificate**. Перед удалением выдается запрос на подтверждение операции.

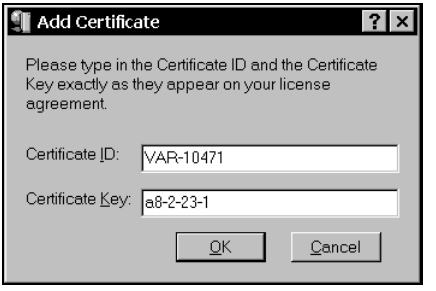


Рис. 28.6. Добавление сертификата к серверу

Управление пользователями

Для каждого сервера есть список пользователей, имеющих право доступа к нему. Список таких пользователей можно просмотреть в правой панели, выбрав в левой панели элемент **Users**. Консоль позволяет добавить или удалить пользователя, а также просмотреть и изменить его атрибуты. Эти действия (кроме удаления) выполняются в окне **User Information** (рис. 28.7), открываемом командой **User Security** или командами контекстного меню списка пользователей.

Рис. 28.7. Атрибуты пользователя

Для пользователя необходимо ввести обязательные атрибуты (**Required Information**), которыми являются его системное имя (**User Name**), пароль (**Password**) и повтор пароля (**Confirm Password**). Имя пользователя также можно выбрать в раскрывающемся списке. Остальные атрибуты являются дополнительными (**Optional Information**) и не

требуют обязательного ввода: имя (**First Name**), отчество (**Middle Name**) и фамилия (**Last Name**).

Управление БД

Список баз данных, зарегистрированных для сервера, отображается в левой панели. Так, на рис. 28.1 для локального сервера (Local Server) такими БД являются `Registration` и `Human Resources`. Зарегистрированная БД может быть подключена или отключена от сервера, что отмечается, соответственно, зеленой галочкой или красным крестом в значке БД.

Управление БД заключается в:

- регистрации БД;
- подключении БД;
- создании и удалении БД;
- просмотре метаданных;
- проверке состояния БД;
- анализе статистики;
- сохранении и восстановлении БД.

Для управления БД используются команды меню **Database** главного окна программы IBConsole, а также команды контекстного меню баз данных и их элементов.

Регистрация базы данных

Регистрация БД начинается командой **Register**, которая открывает окно **Register Database and Connect** (рис. 28.8). В этом окне необходимо указать (выбрать) главный файл БД (**File**), а также псевдоним (имя) БД (**Alias**), под которым она будет зарегистри-

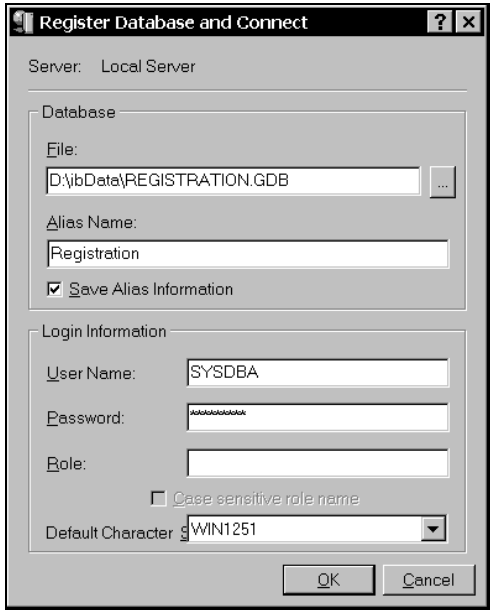


Рис. 28.8. Регистрация БД на сервере

рована на сервере, обозначенном надписью **Server**. По умолчанию в качестве псевдонима БД предлагается имя ее главного файла с расширением.

Остальные данные не являются обязательными. Однако если задать имя и пароль пользователя, то после регистрации выполняется подключение указанной БД.

Исключение БД из списка регистрации сервера выполняется командой **Unregister**, при этом БД предварительно должна быть отключена от сервера. При выполнении операции запрашивается подтверждение.

Подключение базы данных

Подключение БД к серверу и *отключение* от него выполняется командами **Connect** и **Disconnect** соответственно. Команда **Connect As** позволяет подключиться к БД с новыми параметрами, указываемыми в окне **Database Connect** (рис. 28.9).

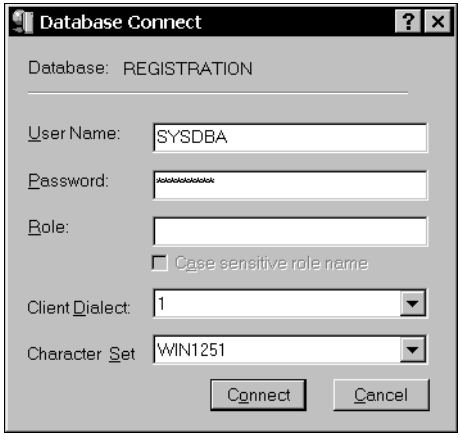


Рис. 28.9. Подключение БД к серверу

Создание базы данных

Консоль позволяет достаточно удобно и быстро *создавать* БД, в том числе многофайловые. Создание БД, а также ее удаление, для которых предназначены команды **Create Database** и **Drop Database**, рассмотрены в *главе 26*.

Просмотр метаданных

Метаданные представляют собой элементы структуры БД. Для выбранной БД их можно просмотреть с помощью команды **View Metadata**, которая открывает окно **Database Metadata**. На рис. 28.10 показаны метаданные БД emloyee.gdb, которая поставляется вместе с сервером InterBase в качестве примера.

Метаданные представляют собой сценарий (скрипт), написанный на языке SQL, который можно распечатать или сохранить в текстовом файле. Сохраненный сценарий впоследствии можно *выполнить*, создав БД со всей ее структурой и данными.

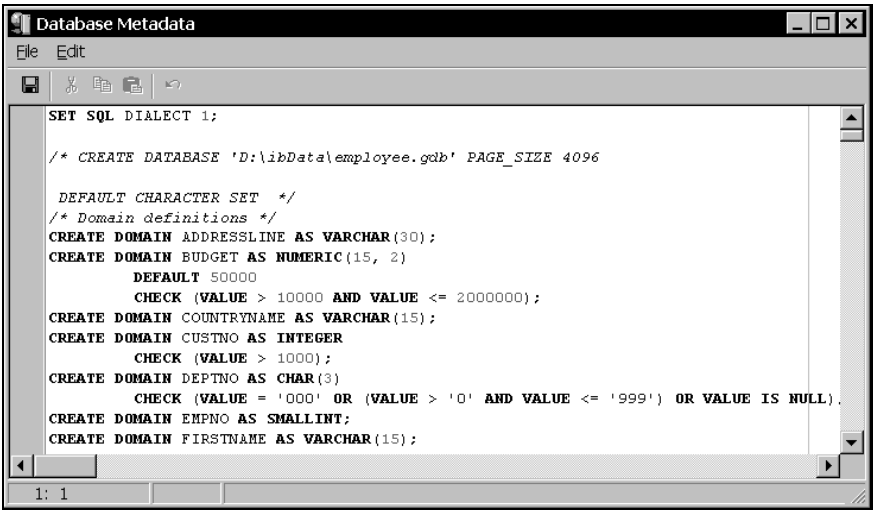


Рис. 28.10. Просмотр метаданных

Сбор мусора

В процессе интенсивной многопользовательской работы в БД накапливается так называемый "мусор", под которым понимают старые версии записей, которые могут образовываться при одновременном доступе к записям нескольких транзакций. Наличие мусора увеличивает размер и фрагментацию БД, поэтому БД надо периодически чистить — "удалять мусор" (удаление мусора также называют "сбором мусора").

Удаление мусора можно выполнять в ручном или автоматическом режимах. В ручном режиме удаление мусора начинается командой **Maintenance | Sweep** (Обслуживание | Чистка). Во втором режиме удаление мусора начинается автоматически, когда общее число примененных к БД транзакций достигает предельного значения. По умолчанию это значение установлено равным 20 000, его можно изменить в поле **Sweep Interval** (Интервал чистки) окна свойств БД (**Database Properties**).

Если при чистке БД работают активные пользователи, то это снижает эффективность удаления мусора, т. к. используемые транзакциями записи не могут быть обработаны "уборщиком". Поэтому удаление мусора следует проводить в периоды наименьшей загрузки БД, например, в ночные часы или в режиме монопольного доступа к ней системного администратора.

Мусор удаляется также при резервном копировании и последующем восстановлении БД.

Проверка состояния базы данных

БД должна находиться в целостном и непротиворечивом состоянии, т. е. содержать правильные данные. Для проверки состояния БД нужно выбрать команду **Maintenance | Validation**, которая открывает диалоговое окно проверки БД **Database Validation** (рис. 28.11).

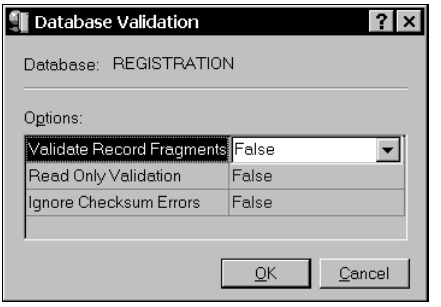


Рис. 28.11. Проверка состояния БД

В надписи **Database** отображается имя проверяемой БД, а группа **Options** позволяет задать параметры поверки:

- **Validate Record Fragments** (проверка структуры БД и структуры страниц);
- **Read Only Validation** (в процессе проверки допускается только читать, но не изменять записи);
- **Ignore Checksum Errors** (ошибки контрольных сумм игнорируются).

По умолчанию все параметры имеют значение `False`, т. е. выключены. После нажатия кнопки **OK** выполняется проверка, о результатах которой выдается соответствующий отчет в окне **Validation Report**.

Анализ статистики

В процессе управления БД собирается определенная информация, характеризующая ее работу и функционирование. Эта информация называется *статистикой*, несмотря на то что часть ее является управляющей информацией, определяемой в том числе и при создании БД (например, размер страницы или дата создания БД). К собственно статистике относятся такие сведения, как частота обновления заголовка БД и гистограмма заполнения страниц.

Для вывода статистики нужно выбрать команду **Maintenance | Database Statistics**, которая открывает диалоговое окно **Database Statistics** (рис. 28.12).

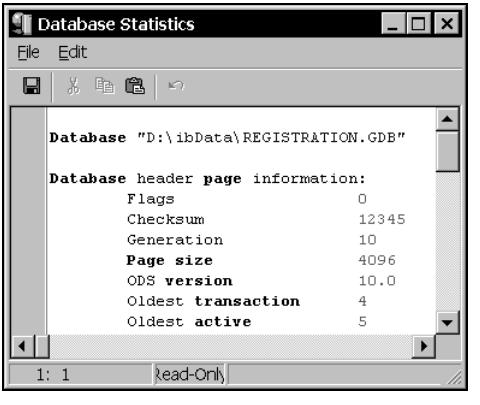


Рис. 28.12. Статистика БД

Выводимые в окне сведения о БД сгруппированы по секциям:

- `Database` (имя БД);
- `Database header page information` (страница заголовка БД):
  - `Flags` (флаги);
  - `Checksum` (контрольная сумма);
  - `Generation` (счетчик обновлений заголовка);
  - `Page size` (размер страницы);
  - `ODS version` (версия формата файла БД);
  - `Oldest transaction` (номер самой старой незавершенной (активной, отмененной или сбойной) транзакции);
  - `Oldest active` (номер самой старой активной транзакции);
  - `Next transaction` (номер, который будет назначен следующей транзакции);
  - `Sequence number` (номер первой страницы);
  - `Next attachment ID` (номер, который будет назначен следующему соединению);
  - `Implementation ID` (идентификатор операционной системы, в которой создана БД);
  - `Shadow count` (число теневых файлов, определенных для БД);
  - `Page buffers` (номер страницы в кэше БД);
  - `Next header page` (номер, который будет назначен следующей странице заголовка);
  - `Creation date` (дата создания БД);
  - `Attributes` (атрибуты БД);
- `Database file sequence` (список файлов БД):
  - `File` (имя файла);
- `Database log page information` (страница журнала БД).

Сохранение и восстановление базы данных

Сохранение БД заключается в создании резервной копии БД, которую впоследствии можно использовать для восстановления данных при сбое. Предварительно БД необходимо отключить.

Для создания резервной копии БД нужно выполнить команду **Maintenance | Backup-Restore | Backup**, которая откроет окно создания резервной копии **Database Backup** (рис. 28.13).

В списке **Alias** группы **Database** выбирается имя сохраняемой БД, а элементы группы **Backup File(s)** определяют файл результата — сохраненной копии БД. В комбинированных списках **Server** и **Alias** выбирается или вводится соответственно имя сервера и БД для сохраняемой БД, а в поле **Filename(s)** вводится полное имя файла копии

БД. В приведенном на рис. 28.13 варианте в имени копии указано слово `Copy`, а расширение gdb оставлено без изменений. В примерах, поставляемых совместно с сервером InterBase, имена копий совпадают с именами исходных файлов, а расширения изменены на gbk.

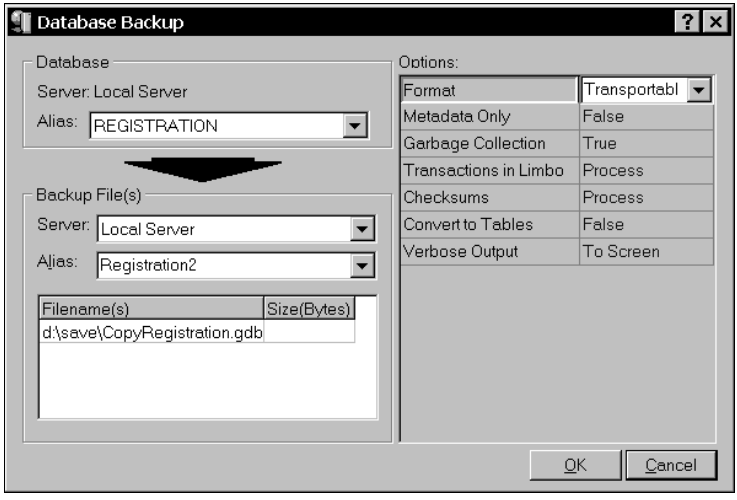


Рис. 28.13. Создание резервной копии БД

Группа **Options** позволяет задать следующие параметры:

- **Format** (Формат) — формат копии, которая по умолчанию создается в переносимом формате, не зависящем от операционной системы: сделанную в этом формате копию можно перенести (восстановить) на компьютер под управлением любой из операционных систем, где установлен сервер InterBase;
- **Metadata Only** (Только метаданные) — копируются только метаданные БД, т. е. ее структура; в результате будет скопирована (создана) пустая БД; по умолчанию параметр имеет значение `False`, поэтому создается полная копия БД;
- **Garbage Collection** (Сбор мусора) — при копировании производится сбор мусора (по умолчанию);
- **Transaction in Limbo** (Транзакция в "забвении") — учитываются сбойные транзакции, которые могут возникнуть при размещении БД на нескольких серверах (значение `Process` по умолчанию); значение `Ignore` предписывает не учитывать сбойные транзакции;
- **Checksums** (Контрольные суммы) — учитываются ошибки, связанные с несовпадением контрольных сумм (значение `Process` по умолчанию); при значении `Ignore` ошибки контрольных сумм игнорируются;
- **Convert to Tables** (Преобразование в таблицы) — преобразование в таблицы, по умолчанию `False`;
- **Verbose Output** (Расширенный вывод) — указывает устройство, на которое в процессе копирования выдается дополнительная информация о ходе процесса, по

умолчанию экран (значение `To Screen`); кроме экрана можно указать файл (`To File`) или отменить выдачу сообщений (`None`).

После завершения процесса выдается отчет о результатах, который в случае успешного создания копии выводится в окне **Database Backup** и имеет следующий вид:

```
gbak: readied database D:\ibData\REGISTRATION.GDB for backup
gbak: creating file d:\save\CopyRegistration.gdb
gbak: starting transaction
gbak: database D:\ibData\REGISTRATION.GDB has a page size of 4096 bytes.
gbak: writing domains
gbak: writing shadow files
gbak: writing tables
gbak: writing functions
...
gbak: writing privilege for user SYSDBA
gbak: writing privilege for user PUBLIC
gbak: writing table constraints
gbak: writing referential constraints
gbak: writing check constraints
gbak: writing SQL roles
gbak: closing file, committing, and finishing. 512 bytes written
```

При наличии ошибок копия не создается, а отчет содержит сообщения об ошибках.

Замечание

Если резервное копирование выполняет не системный администратор, а пользователь, то ошибки могут быть связаны с отсутствием у него прав доступа к информации БД.

Резервная копия представляет собой архивный файл, размер которого в несколько раз меньше, чем размер исходного файла (файлов) БД.

Процесс восстановления БД из резервной копии БД инициируется командой **Maintenance | Backup-Restore | Restore**, открывающей диалоговое окно **Database Restore** (рис. 28.14).

В списке **Alias** группы **Backup File(s)** выбирается имя сохраненной БД, после чего имя ее файла-копии автоматически выводится в поле **Filename(s)**. Если имя БД отсутствует в списке, то можно задать ее файл, выбрав в списке элемент **File** и указав его имя в открывшемся окне **Open**. Элементы группы **Database** определяют сервер (**Server**), имя БД (**Alias**) и главный файл БД (**Filename(s)**).

Группа **Options** содержит следующие параметры восстановления:

- **Page Size (Bytes)** — размер в байтах страницы восстанавливаемой БД (по умолчанию 1024);
- **Overwrite** (Перезаписать) — если расположение и имя восстанавливаемой копии совпадают с расположением и именем существующей БД, то последняя будет заменена копией; по умолчанию имеет значение `False`, т. е. сохраняется существующая БД;

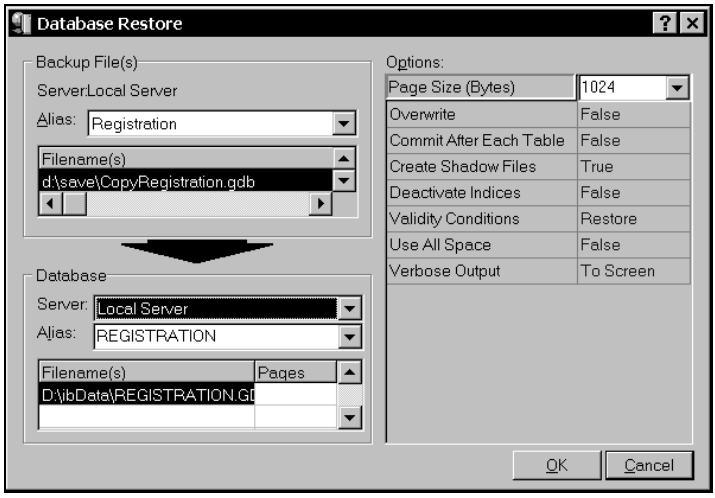


Рис. 28.14. Восстановление БД

- **Commit After Each Table** (Подтверждение после каждой таблицы) — при восстановлении каждой таблицы выдается запрос на подтверждение этой операции; по умолчанию этот параметр имеет значение `False`, и запрос не выдается;
- **Create Shadow Files** (Создание теневых файлов); по умолчанию `False`, т. е. восстановление выполняется без создания теневой (зеркальной) копии БД;
- **Deactivate Indices** (Отключить индексы) — восстановление выполняется с отключенными индексами; по умолчанию этот параметр имеет значение `False`, и индексы активны;
- **Validity Conditions** (Условия проверки) — при значении `Restore` (по умолчанию) выполняется восстановление ограничений ссылочной целостности, при значении `Ignore` ограничения не восстанавливаются;
- **Use All Space** (Использовать все имеющееся пространство); по умолчанию `False`;
- **Verbose Output** (Расширенный вывод) — указывает устройство, на которое в процессе копирования выдается дополнительная информация о ходе процесса, по умолчанию экран (значение `To Screen`); кроме экрана, можно указать файл (`To File`) или отменить выдачу сообщений (`None`).

После завершения процесса отчет о результатах, который в случае успешного восстановления БД выводится в окне **Database Restore** и имеет следующий вид:

```
gbak: opened file d:\save\CopyRegistration.gdb
gbak: transportable backup -- data in XDR format
gbak:     backup file is compressed
gbak: Reducing the database page size from 4096 bytes to 1024 bytes
gbak: created database D:\ibData\REGISTRATION.GDB, page_size 1024 bytes
gbak: started transaction
gbak:     restoring privilege for user SYSDBA
gbak:     restoring privilege for user SYSDBA
gbak:     restoring privilege for user SYSDBA
```

```
gbak:     restoring privilege for user SYSDBA
gbak:     restoring privilege for user SYSDBA
gbak:     restoring privilege for user PUBLIC
gbak: creating indexes
gbak: finishing, closing, and going home
```

Интерактивное выполнение SQL-запросов

Консоль IBConsole позволяет в интерактивном режиме выполнять инструкции, заданные на языке SQL. Выполнение SQL-запросов и получение их результатов выполняются в окне **Interactive SQL** (рис. 28.15), открываемом командой **Tools | Interactive SQL**. Это окно реализует практически ту же функциональность, которая в предыдущих версиях сервера InterBase была реализована в программе Windows Interactive SQL (WISQL). В заголовке окна отображается имя файла БД, а в строке состояния — его полное имя. Одновременно можно открыть несколько окон **Interactive SQL** для различных БД.

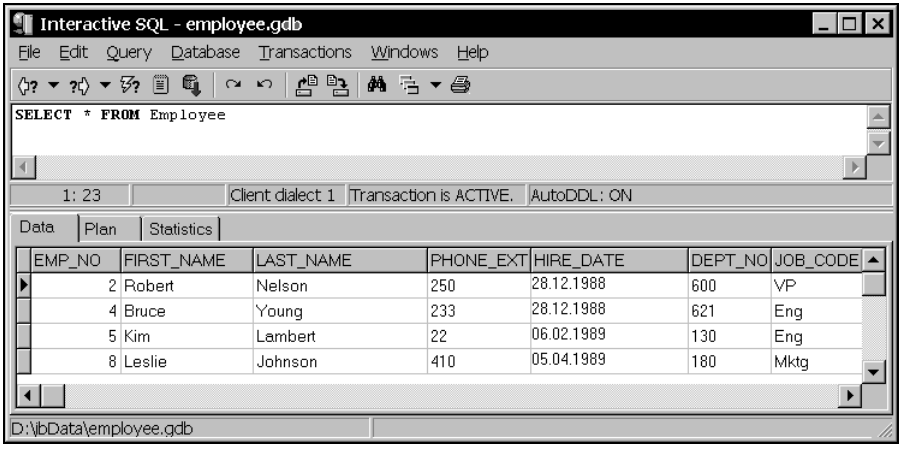


Рис. 28.15. Окно интерактивного выполнения SQL-запросов

В окне **Interactive SQL** можно выполнять различные операции с БД, включая создание и удаление БД и ее таблиц, соединение с БД, просмотр и изменение данных. Выполнение операций с БД основано на выполнении соответствующих инструкций языка SQL (SQL-запросов). Эти инструкции формируются и выполняются автоматически при выборе определенных команд меню. Так, при создании БД на основании указанных параметров формируется инструкция `CREATE DATABASE`.

Кроме того, можно набирать и выполнять инструкции вручную. Инструкции вводятся в верхней части окна, а в нижней его части выдаются результаты их выполнения.

Инструкции можно набирать и выполнять поочередно. Перемещение между отдельными инструкциями осуществляется нажатием кнопок с изображением желтых стрелок. Кнопка со стрелкой влево выводит в окне предыдущую, а со стрелкой вправо — следующую инструкцию. Выполнение инструкции осуществляется при нажатии кнопки с изображением желтой молнии.

Результаты работы инструкции можно запомнить в обычном текстовом файле, выбрав команду сохранения результата **Query | Save Output**.

Последовательность действий, заданных после запуска окна **Interactive SQL**, можно сохранить в виде "истории". Для этого предназначена команда **File | Save SQL History**. "История" запоминается в виде текстового файла с расширением hst или txt.

Последовательность инструкций SQL представляет собой сценарий, или скрипт, его удобно использовать для автоматизации операций с БД. Сохранение сценария выполняется командой **Query | Save Script**. Скрипт запоминается в виде текстового файла с расширением sql или txt.

Сохраненный сценарий впоследствии можно загрузить, для этого надо выбрать команду **Query | Load Script** и указать имя сценария.

При выполнении операций, связанных с изменением БД, автоматически используется механизм транзакций, т. е. одновременно с началом изменяющей БД операции запускается транзакция. Такой режим запуска транзакций, как вы уже знаете, называется неявным. Для фиксации выполненных изменений нужно выполнить оператор `COMMIT`, а для их отмены (отката) — `ROLLBACK` (это же можно сделать с помощью команд **Transactions | Commit** и **Transactions | Rollback**).

Параметры выполнения SQL-запросов устанавливаются в окне **SQL Options**, открываемом командой **Edit | Options** (рис. 28.16).

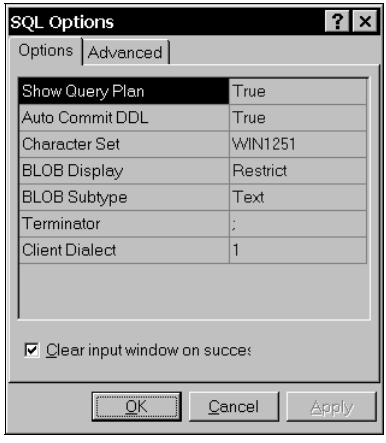


Рис. 28.16. Установка параметров выполнения SQL-запросов

На странице **Options** можно задать следующие параметры:

- **Show Query Plan** (Показывать план выполнения запроса);
- **Auto Commit DDL** (Автоматическое подтверждение операций DDL) — инструкции определения данных, например создания таблиц, автоматически подтверждаются, т. е. связанная с инструкциями неявная транзакция не требует дополнительного подтверждения;

- **Character Set** (Набор символов, используемый для кодировки строк); в нашей стране рекомендуется задавать набор `WIN1251`;
- **BLOB Display** (Режим отображения BLOB-объектов); может принимать значения `Restrict`, `Enabled` и `Disabled`;
- **BLOB Subtype** (Тип данных, содержащихся в объекте BLOB); по умолчанию `Text`;
- **Terminator** (Разделитель, используемый в качестве знака окончания SQL-запроса) — по умолчанию точка с запятой (`;`);
- **Client Dialect** (Диалект (версия) языка SQL).

Флажок **Clear input window on success** управляет режимом очистки верхней половины окна с текстом SQL-запроса в случае его успешного выполнения. По умолчанию флажок установлен, и окно очищается.

Замечание

В коде хранимых процедур и триггеров в качестве разделителя используется знак `;`, который не является окончанием SQL-запроса. Поэтому при их отладке для параметра **Terminator** необходимо установить другое значение, например пробел.

Дополнительные параметры устанавливаются на странице **Advanced** (рис. 28.17). Наибольший интерес представляет группа переключателей **Transactions**, которые определяют режим подтверждения незавершенной транзакции при прекращении работы с БД: подтверждать незавершенную транзакцию (**Commit on exit**) или отменять ее (**Rollback on exit**).

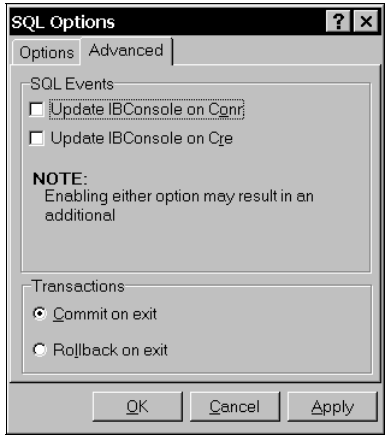


Рис. 28.17. Установка дополнительных параметров выполнения SQL-запроса

Управление БД в окне **Interactive SQL** фактически происходит в ручном режиме и заключается во вводе и выполнении инструкций SQL. Единственным средством автоматизации, например, при создании таблиц или триггеров является использование сценариев, хотя и в этом случае сценарии должны быть предварительно подготовлены. По удобству работы окно **Interactive SQL** уступает даже относительно простым программам типа Database Desktop. Несмотря на это, управление БД рассмотрено нами именно

с использованием интерактивного SQL. Это сделано для того, чтобы показать особенности работы с удаленными БД, а также продемонстрировать язык SQL для удаленного сервера.

Кроме консоли IBConsole и подобных ей программ, есть так называемые средства CASE, с помощью которых разработчик может управлять БД в интерактивном режиме.

Программа SQL Monitor

Программа SQL Monitor представляет собой инструмент, позволяющий отслеживать выполнение SQL-запросов к удаленным БД. Для запуска программы нужно выбрать команду **Database | SQL Monitor** или запустить файл sqlmon.exe, находящийся в каталоге BIN главного каталога Delphi. Программу также можно запустить через меню **Пуск** Windows командой **Программы | Borland Delphi 7 | SQL Monitor**.

Программа SQL Monitor (Монитор) отслеживает операции доступа к удаленным БД с помощью драйверов SQL-Links. Поэтому для локальных БД применять эту программу бесполезно, даже если используются средства языка SQL. Это связано с тем, что для локальных БД применяются локальные драйверы.

Замечание

Если в приложении использованы компоненты страницы **InterBase** Палитры компонентов, например, набор данных `IBQuery`, то доступ к БД выполняется напрямую через BDE без использования драйвера SQL-Links. В этом случае контролировать выполнение SQL-запросов к БД InterBase с помощью программы SQL Monitor невозможно.

После запуска Монитор автоматически отслеживает порядок выполнения SQL-запросов (выполняет трассировку инструкций), ведя журнал работы с удаленной БД. Монитор отслеживает запросы для указанного в команде **Clients** клиента, при отладке Delphi-приложения таким клиентом является Delphi 7. Строки журнала выводятся в верхней части окна программы (рис. 28.18). В нижней части окна отображается строка последнего запроса.

Для сохранения журнала в текстовом файле нужно выполнить команду **File | Save Log** и указать имя файла, для очистки журнала — команду **Edit | Clear**, для копирования его содержимого в буфер — команду **Edit | Copy**.

Параметры Монитора устанавливаются в окне параметров трассировки **Trace Options** (рис. 28.19), открываемом командой **Options | Trace Options**.

С помощью параметров Монитора можно регулировать степень подробности информации, заносимой в журнал. Монитор имеет следующие параметры:

- **Prepared Query Statements** (подготовленные запросы, передаваемые на сервер);
- **Executed Query Statements** (выполненные на сервере запросы);
- **Input Parameters** (входные параметры) — данные, передаваемые на сервер в качестве параметров запросов;
- **Fetched Data** (данные, возвращаемые сервером);
- **Statement Operations** (операции с запросами) — `ALLOCATE`, `PREPARE`, `EXECUTE` и `FETCH`;

- **Connect/Disconnect** (операции соединения с сервером и отключения от него);
- **Transactions** (операции управления транзакциями);
- **Blob I/O** (ввод/вывод данных типа BLOB);
- **Miscellaneous** (остальные операции);
- **Vendor Errors** (сообщения об ошибках, возвращаемые сервером);
- **Vendor Calls** (вызовы API-функций сервера).

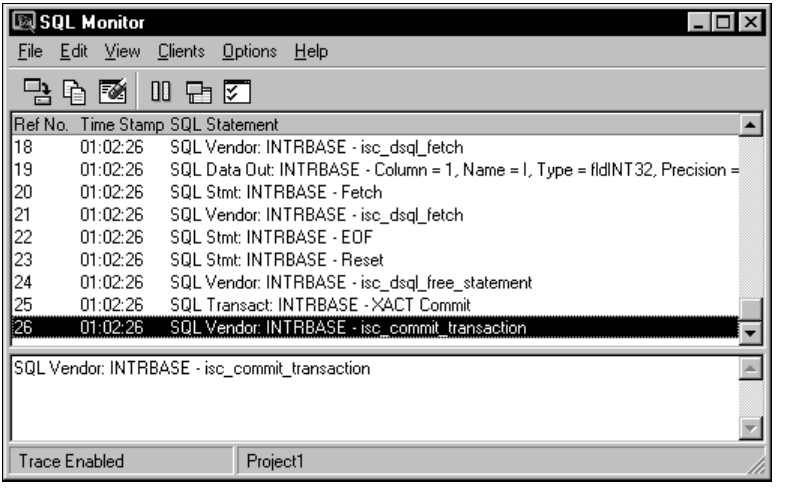


Рис. 28.18. Окно программы SQL Monitor

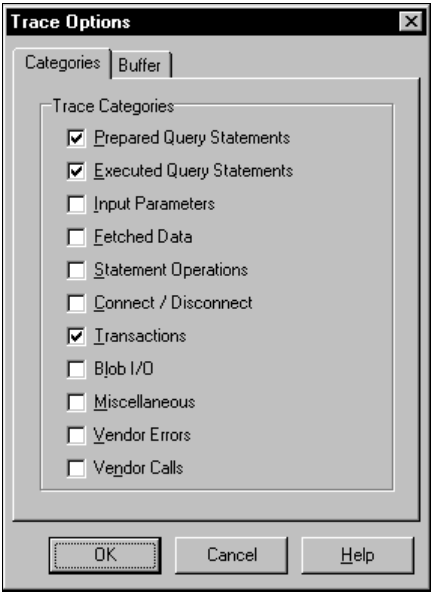


Рис. 28.19. Диалоговое окно установки параметров трассировки

По умолчанию включены все параметры, и отслеживание выполнения запроса осуществляется максимально подробно. При этом даже простой запрос приводит к появлению в журнале значительного числа строк. Например, после выполнения запроса на отбор записей

```
SELECT * FROM ti
```

в журнал будет занесено 26 строк, что не облегчает, а скорее затрудняет проверку выполнения запроса. Поэтому в данном случае можно ограничиться только следующими параметрами:

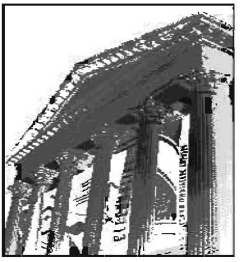
- **Prepared Query Statements**;
- **Executed Query Statements**;
- **Transactions**.

Тогда после выполнения приведенного запроса в журнал будут занесены такие данные:

```
1       01:10:47  SQL Prepare: INTRBASE — SELECT I FROM ti
2       01:10:47  SQL Transact: INTRBASE — XACT (UNKNOWN)
3       01:10:47  SQL Execute: INTRBASE — SELECT I FROM ti
4       01:10:47  SQL Transact: INTRBASE — XACT Commit
```

Монитор можно разместить поверх всех окон командой **Options | Always on Top**.

ГЛАВА 29



Трехуровневые приложения

Развитие архитектуры "клиент-сервер" привело к появлению трехуровневой архитектуры, в которой кроме сервера и приложений-клиентов (клиентов) дополнительно присутствует сервер приложений. *Сервер приложений* является промежуточным уровнем, обеспечивающим организацию взаимодействия клиентов ("тонких" клиентов) и сервера, например, выполнение соединения с сервером, разграничение доступа к данным и реализацию бизнес-правил. Сервер приложений реализует работу с клиентами, расположенными на различных платформах, т. е. функционирующими на компьютерах различных типов и под управлением различных операционных систем. Сервер приложений также называют *брокером данных* (broker — посредник).

Основные достоинства трехуровневой архитектуры "клиент-сервер":

- снижение нагрузки на сервер;
- упрощение клиентских приложений;
- единое поведение всех клиентов;
- упрощение настройки клиентов;
- независимость от платформы.

Информационные системы, основанные на трехуровневой сетевой архитектуре, называют также *распределенными*.

Принципы построения трехуровневых приложений

В системе Delphi постоянно совершенствуются используемые технологии программирования и соответствующие им наборы компонентов. Так, в предыдущих версиях системы применяемые при разработке многоуровневых приложений технологии и средства объединялись под общим названием MIDAS (Multi-tier distributed application services — службы многоуровневых распределенных приложений). В последней версии системы соответствующие средства (компоненты и объекты) получили общее название DataSnap.

В Delphi 7 поддерживается создание многоуровневых приложений, основанных на перечисленных далее технологиях межпрограммного и межкомпьютерного взаимодействия.

- Модель DCOM (Distributed Component Object Model — модель распределенных компонентных объектов) позволяет использовать объекты, расположенные на другом компьютере.
- Сервер MTS (Microsoft Transaction Server — сервер транзакций Microsoft) является дополнением к технологии COM, разработанной фирмой Microsoft, и предназначен для управления транзакциями.
- Модель COM+ (Component Object Model+ — усовершенствованная объектная модель компонентов) фирмы Microsoft введена в Windows 2000 и интегрирует технологии MTS в стандартные службы COM.
- Сокеты TCP/IP (Transport Control Protocol/Protocol Internet — транспортный протокол/протокол Интернета) используются для соединения компьютеров в различных сетях, в том числе в Интернете.
- CORBA (Common Object Request Broker Architecture — общедоступная архитектура с брокером при запросе объекта) позволяет организовать взаимодействие между объектами, расположенными на различных платформах.
- SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) служит универсальным средством обеспечения взаимодействия с клиентами и серверами Web-служб на основе кодирования XML и передачи данных по протоколу HTTP.

При создании трехуровневого приложения разработка БД и использование сервера принципиально не отличаются от уже рассмотренного случая двухуровневых приложений. Главные особенности трехуровневого приложения связаны с созданием сервера приложений и клиентского приложения, а также с организацией взаимодействия между ними. Для разработки многоуровневых приложений, кроме рассмотренных ранее средств, используются удаленные модули данных и компоненты, размещенные на странице **DataSnap** Палитры компонентов.

В трехуровневой архитектуре с использованием механизма доступа BDE процессор баз данных в обязательном порядке устанавливается совместно с сервером приложений, при этом на клиентском компьютере должна быть установлена только библиотека DBClient.dll относительно небольшого размера (210 Кбайт). Таким образом, на компьютере пользователя "тонким" является не только клиент, но и процессор баз данных. Соответствующая трехуровневая архитектура схематично представлена на рис. 29.1. В частном случае два или все три уровня могут располагаться на одном компьютере, что широко используется при отладке приложений.

Взаимодействие между сервером приложений и клиентом организуется через интерфейс провайдера, называемый *интерфейсом оператора* или просто *провайдером*. Этот интерфейс обеспечивает передачу информации в виде *пакетов данных*. Физически пакеты данных представляют собой совокупности двоичных кодов, образующих блоки. Логически пакет данных является подмножеством набора данных, которое содержит данные записей, а также метаданные (информация об именах и типах полей) и ограни-

чения. Провайдер обеспечивает разбиение данных на пакеты, а также кодирование пакетов в зависимости от используемого сетевого протокола.

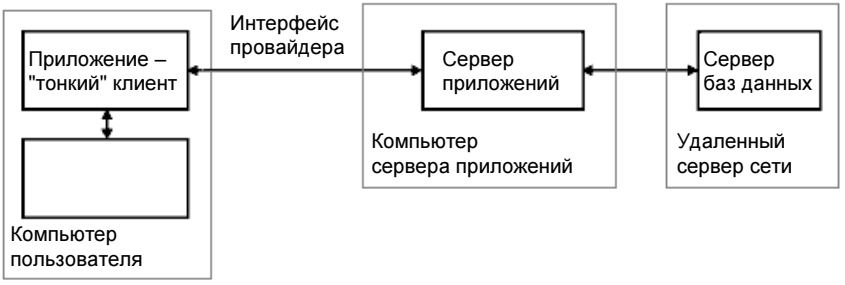


Рис. 29.1. Трехуровневая архитектура типа "клиент-сервер"

Данные, отредактированные клиентом, пересылаются обратно в так называемых *дельта-пакетах*. В дельта-пакете содержится информация о старых и новых значениях записей. Перед внесением в записи БД требуемых изменений (т. е. перед пересылкой записей серверу БД) сервер приложений выполняет проверку корректности и допустимости изменений. Изменения могут быть отвергнуты, например, если запись уже была изменена другим пользователем. В этом случае сервер приложений генерирует событие `OnReconcilError`, которое используется для распознавания и обработки конфликтов между клиентами.

Сервер приложений

Сервер приложений создается на основе удаленного модуля данных, который служит для размещения компонентов, а также для обеспечения взаимодействия с сервером и клиентами. Для создания различных серверов приложений предназначены следующие разновидности удаленных модулей данных:

- **Remote Data Module** — для серверов DCOM, TCP/IP и OLEnterprise;
- **Transactional Data Module** — для сервера MTS;
- **WebSnap Data Module** — для сервера Web;
- **SOAP Server Data Module** — для серверов SOAP.

Удаленные модули данных расположены на страницах **Multitier**, **WebSnap** и **WebServices** Хранилища объектов. В удаленном модуле данных размещаются те же компоненты, что и в простом модуле данных, например, для механизма доступа с помощью BDE такие компоненты, как `Query`, `Database`, `Session`, предназначенные для организации доступа к данным.

Рассмотрим создание простейшего сервера приложений — сервера DCOM, взаимодействие с которым основано на технологии DCOM. Для работы этого сервера необходимо, чтобы в системе была установлена программная поддержка функционирования распределенных COM-объектов, которая имеется в операционных системах Windows 98/NT/2000. Для Windows 95 ее нужно устанавливать отдельно. Поддержка распределенных COM-объектов устанавливается автоматически при инсталляции ряда про-

грамм Windows, кроме того, соответствующие средства можно загрузить из Интернета по адресу:

http://www.microsoft.com/com/dcom95/download-f.htm

Добавление к проекту удаленного модуля данных выполняется выбором объекта **Remote Data Module** страницы **Multitier** Хранилища объектов. При добавлении модуля выводится диалоговое окно мастера **Remote Data Module Wizard**, в котором нужно задать параметры модуля (рис. 29.2).

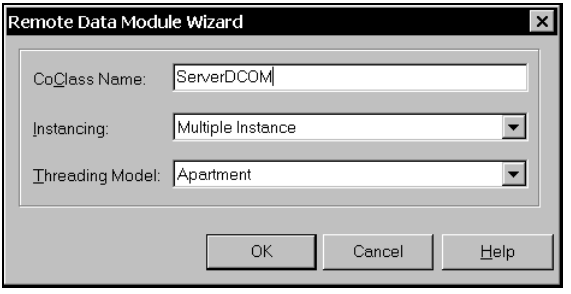


Рис. 29.2. Добавление удаленного модуля данных

В поле редактирования **CoClass Name** вводится имя модуля данных.

В списке **Instancing** (Создание экземпляров) выбирается способ запуска модуля:

- **Internal** — экземпляр модуля данных создается на сервере в случае, когда модуль данных является частью библиотеки DLL;
- **Single Instance** — для каждого клиента в его адресном пространстве создается один экземпляр удаленного модуля данных, и каждое клиентское соединение запускает этот свой экземпляр;
- **Multiple Instance** — один экземпляр приложения (процесс) представляет все удаленные модули данных, созданные для клиентов (по умолчанию); каждый удаленный модуль данных предназначен для одного клиентского соединения, но все они разделяют одно и то же адресное пространство.

В списке **Threading Model** (Потоковая модель) выбирается способ вызова интерфейса клиента, если модуль данных является частью библиотеки DLL:

- **Single** — библиотека получает запросы клиента по одному;
- **Apartment** — одновременно обрабатывается несколько запросов клиентов, для каждого из которых создан отдельный экземпляр модуля данных (по умолчанию);
- **Free** — отдельный экземпляр модуля данных одновременно может отвечать на несколько запросов клиентов;
- **Both** — отдельный экземпляр модуля данных одновременно может отвечать на несколько запросов клиентов, результаты обработки также возвращаются одновременно;
- **Neutral** — разные клиенты могут одновременно вызывать удаленный модуль данных из нескольких потоков, при этом модель COM следит за тем, чтобы не было

конфликта вызовов (однако нужно иметь в виду возможный конфликт потоков: он отслеживается только в версии COM+, при отсутствии ее используется потоковая модель типа Apartment).

После нажатия кнопки **OK** модуль данных с установленными параметрами добавляется к проекту. В приведенном на рис. 29.2 примере модулю присвоено имя `ServerDCOM`, а два других параметра оставлены без изменений.

На этапе проектирования внешний вид удаленного модуля не отличается от вида простого модуля данных, рассмотренного в *главе 16*, посвященной созданию информационной системы. Как и в простом модуле, в удаленном модуле данных размещаются невизуальные компоненты, используемые для доступа к данным. Здесь могут быть наборы компонентов для механизмов доступа ADO, BDE, dbExpres и InterBase Express. К примеру, для технологии BDE чаще всего этими компонентами являются рассмотренные ранее `Query`, `Table`, `Database`, `Session`, а также провайдер `DataSetProvider`. В самом простом случае достаточно разместить в модуле только набор данных. Например, разместим в удаленном модуле набор данных `Query` и зададим для него значения свойств `DataBaseName` и `SQL` так, чтобы включить в набор все поля всех записей таблицы `Personnel`. Указанным свойствам присвоим значения:

- свойству `DataBaseName` — значение `BDPlace`;
- свойству `SQL` — значение `SELECT * FROM Personnel.db`.

На этом создание простейшего сервера DCOM закончено. Перечислим еще раз действия, которые были при этом выполнены:

- к проекту добавлен удаленный модуль данных;
- в модуле размещен компонент набора данных и присвоены значения его свойствам.

Созданное приложение сервера состоит из следующих частей:

- проект;
- главная форма приложения;
- удаленный модуль данных;
- модуль библиотеки типов.

Разработка проекта и главной формы приложения не имеют принципиальных отличий от разработки обычного приложения Delphi. Отметим, что для сервера приложений основная функциональная нагрузка приходится на удаленный модуль данных. В главной форме можно разместить вспомогательные компоненты и выполнять некоторые сервисные действия, например, вести подсчет клиентов, подключенных к серверу, и выводить показания этого счетчика в надписи `Label`, размещенной в главной форме сервера.

Библиотека типов создается автоматически, а ее модуль сохраняется на диске при сохранении других файлов проекта. Библиотека занимает два файла: *Project*.tlb и *Project*_TLB.pas, где *Project* — имя проекта.

После создания сервера DCOM его нужно зарегистрировать как сервер автоматизации. Регистрация сервера выполняется системой Windows автоматически при запуске приложения сервера.

По умолчанию интерфейс провайдера обеспечивает набор данных, в нашем случае это `Query1`. Кроме того, Delphi включает в свой состав компонент `DataSetProvider`, который предоставляет большие возможности по управлению интерфейсом провайдера, включая обмен XML-данными.

Простейший сервер DCOM представляет собой *удаленный брокер данных*, который обеспечивает соединение с сервером БД и передачу данных клиенту и обратно. Для расширения функциональности сервера приложений к нему добавляются бизнес-правила, предназначенные для поддержания БД в целостном состоянии и реализующие ограничения, применяемые к данным.

Поддержка механизма ограничений обеспечивается *брокером ограничений*. Для набора данных и его отдельных полей можно задавать ограничения на значения полей не только в приложении клиента, но и в сервере приложений (удаленном модуле данных). Ограничения, заданные в сервере приложений, пересылаются клиенту вместе с данными в пакете данных, и эти ограничения действуют наряду с ограничениями, заданными в приложении клиента.

Для реализации ограничений в сервере приложений при использовании технологии BDE можно использовать свойство `Constraints` типа `TCheckConstraints` наборов данных `Table` и `Query`. Тип `TCheckConstraints` представляет собой коллекцию (набор) отдельных ограничений типа `TCheckConstraint`, имеющих следующие свойства:

- `CustomConstraint` типа `String` — код SQL, описывающий ограничение;
- `ErrorMessage` типа `String` — текст, выдаваемый пользователю при нарушении данного ограничения;
- `FromDictionary` типа `Boolean` — признак, значение `True` которого указывает, что ограничение выбирается из словаря данных; по умолчанию свойство имеет значение `False`, и словарь данных не используется;
- `ImportedConstraint` типа `String` — код SQL, описывающий ограничение, которое импортировано из словаря данных.

Для задания ограничений нужно выделить набор данных и в окне Инспектора объектов щелчком в области значения свойства `Constraints` открыть окно, показанное на рис. 29.3, справа. Центральную часть окна занимает список ограничений, применяемых к набору данных, имя которого выводится в заголовке окна (на рисунке — `Query1`). Добавление к списку нового ограничения выполняется командой **Add** контекстного меню, нажатием клавиши <Insert> или нажатием крайней левой кнопки на панели инструментов. Существующие ограничения можно удалять и перемещать в пределах списка, эти действия выполняются с помощью команд контекстного меню, нажатия клавиш или кнопок панели инструментов.

Сразу после добавления ограничение "пустое", и в списке выводится название его типа `TCheckConstraint` (на рис. 29.3 это третье ограничение). Для задания ограничения нужно его описать, например, присвоив значения свойствам `CustomConstraint` и `ErrorMessage`. После того как свойство `CustomConstraint` получит значение, оно будет выведено в списке ограничений. Свойства ограничения становятся доступными через Инспектор объектов после выбора ограничения в списке.

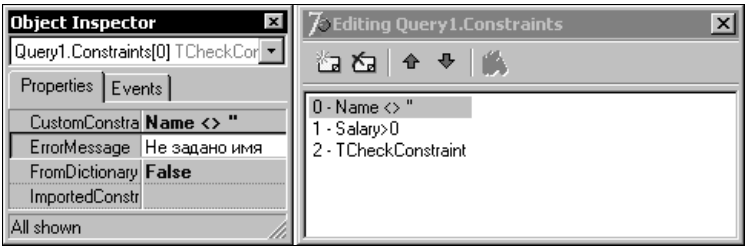


Рис. 29.3. Определение ограничений для набора данных `Query1`

В приведенном на рис. 29.3 примере для данных о сотрудниках организации (таблица `Personnel`) установлены ограничения на значения полей `Name` и `Salary`: поле имени не может быть пустым, а значение оклада должно быть положительным. При нарушении этих ограничений пользователю выдаются соответствующие сообщения: например, если не задано значение поля `Name`, то выдается сообщение `Не задано имя!`.

Как уже было сказано, ограничения сервера приложений действуют в дополнение к ограничениям, заданным в приложении клиента. Так обеспечивается распределение ограничений, применяемых к данным, между отдельными уровнями информационной системы. Достоинством размещения бизнес-правил на сервере приложений является то, что они одинаковы для всех клиентов и что облегчаются внесение изменений в информационную систему и ее настройка.

Свойства `CustomConstraint`, `ConstraintErrorMessage` и `ImportedConstraint` объектов типа `TField` позволяют задать ограничения для отдельных полей набора данных. Эти свойства аналогичны свойствам `CustomConstraint`, `ErrorMessage` и `ImportedConstraint` объекта типа `TCheckConstraints`.

Приложение клиента

Приложение "тонкого" клиента отличается от ранее рассмотренного приложения "толстого" клиента в первую очередь тем, что для "тонкого" клиента нужно выполнить следующие действия:

- организовать связь между приложением клиента и сервером приложений;
- обеспечить обмен информацией между наборами данных клиента и сервера.

Для этого используются компоненты соединения и клиентский набор данных `ClientDataSet`, размещаемые в форме клиента.

Выбор компонента, используемого для соединения с сервером приложений, зависит от типа сервера (типа коммуникационного протокола):

- `DCOMConnection` — для соединения с помощью DCOM;
- `SocketConnection` — для соединения через сокеты TCP/IP;
- `WebConnection` — для соединения с помощью HTTP;
- `SOAPConnection` — для соединения с помощью SOAP (HTTP и XML).

Создадим приложение клиента, подключаемого к рассмотренному выше серверу DCOM, для чего разместим в главной форме компонент `DCOMConnection` со страницы **DataSnap** Палитры компонентов. Основные свойства этого компонента:

- `ComputerName` типа `String` (имя компьютера, на котором расположен сервер приложений);
- `ServerName` типа `String` (имя сервера приложений);
- `ServerGUID` типа `String` (универсальный уникальный идентификатор GUID сервера приложений);
- `Connected` типа `Boolean` (признак, управляющий активностью соединения).

Для указания компьютера, на котором расположен сервер приложений, удобно использовать окно **Browse for Computer** (рис. 29.4), открываемое через Инспектор объектов. После выбора сетевого компьютера и нажатия кнопки **OK** имя выбранного компьютера присваивается в качестве значения свойству `ComputerName`. Если сервер расположен на одном компьютере с приложением клиента (что удобно при отладке приложений), то значение свойства `ComputerName` не задается.

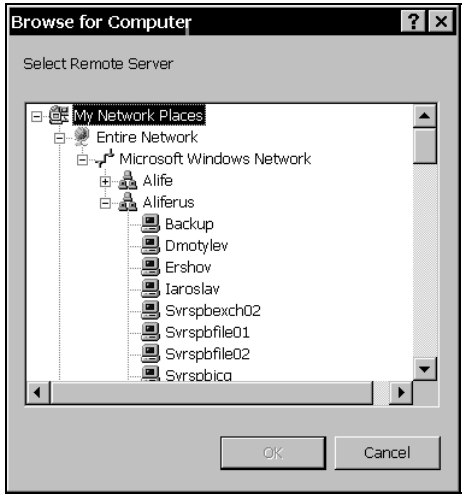


Рис. 29.4. Выбор сетевого компьютера

После того как компьютер задан, имена доступных (зарегистрированных) серверов автоматизации можно выбирать с помощью Инспектора объектов в списке значений свойства `ServerName`. Имя сервера является составным и включает в себя имя проекта приложения сервера и имя модуля данных, задаваемое для удаленного модуля данных, например, `Server.ServerDCOM`.

Задание имени сервера приводит к автоматической установке для выбранного сервера идентификатора GUID, который присваивается в качестве значения свойству `ServerGUID`. GUID (Globally Unique Identifier — глобальный уникальный идентификатор) представляет собой 128-битную константу, присваиваемую объекту COM для его однозначной идентификации. Значение GUID отображается в модуле библиотеки

типов сервера приложений. Сервер можно также задать, установив значение свойству `ServerGUID`, в этом случае значение свойства `ServerName` заполняется автоматически. Однако первый путь, связанный с выбором имени сервера, более удобен.

Чтобы протестировать соединение с сервером приложений, свойство `Connected` устанавливается в значение `True`. В этом случае сервер запускается автоматически, и с ним устанавливается соединение. В общем случае значение этого свойства можно не трогать, т. к. оно автоматически устанавливается в `True` при выборе провайдера для клиентского набора данных `ClientDataSet`.

Клиентский набор данных `ClientDataSet` предназначен для работы с записями, поступающими с сервера приложений. Перечислим свойства этого компонента:

- `RemoteServer` типа `TCustomRemoteServer` (соединение, используемое для связи с сервером);
- `ProviderName` типа `String` (провайдер, обеспечивающий передачу данных);
- `Active` типа `Boolean` (признак, указывающий, открыт или закрыт набор данных);
- `PacketRecords` типа `Integer` (размер пакета данных);
- `FileName` типа `String` (имя файла для обмена данными с диском).

В качестве значения свойства `RemoteServer` можно указывать любой из компонентов, используемых для соединения с сервером: `DCOMConnection`, `SocketConnection`, `WebConnection`, `SOAPConnection`. Нужное значение удобно выбирать в списке Инспектора объектов. Поскольку для соединения с сервером DCOM в форме размещен компонент `DCOMConnection`, в списке нужно выбрать его имя `DCOMConnection1`, присвоенное компоненту по умолчанию.

После того как соединение выбрано, с помощью свойства `ProviderName` задается провайдер, обеспечивающий передачу данных клиенту. При раскрытии в окне Инспектора объектов списка значений этого свойства автоматически запускается сервер приложений, если он еще не был запущен, обеспечивая выдачу клиенту списка доступных провайдеров (наборов данных). Выбор имени провайдера приводит к соединению клиентского набора данных `ClientDataSet` с соответствующим набором данных сервера.

В удаленном модуле данных рассмотренного ранее сервера приложений расположен набор данных `Query1`, предоставляющий интерфейс `Iprovider`, имя которого можно выбрать в качестве значения, присваиваемого свойству `ProviderName`.

Для работы с данными в приложении клиента размещаются визуальные компоненты и источник данных `DataSource`, которые связываются между собой, а также с клиентским набором данных аналогично тому, как это выполнялось для рассмотренных ранее локальных приложений и для приложений "толстого" клиента. В приведенном на рис. 29.5 примере в форме приложения размещены компоненты `DBGrid` (сетка) и `DataSource`, свойство `DataSet` которого имеет значение `ClientDataSet1` (имя компонента `ClientDataSet` по умолчанию). После установки свойства `Active` клиентского набора данных в значение `True` в сетке отображаются записи набора данных `Personnel`, отбор которых обеспечивает инструкция `SELECT` набора данных `Query1` сервера приложений.

Чтобы расширить функциональность рассмотренного простейшего "тонкого" клиента, к нему добавляют возможности, связанные с модификацией записей, передачей изме-

нений на сервер приложений (записью их в БД), обработкой конфликтов между клиентами, а также с выполнением ряда других действий.

Рис. 29.5. Форма приложения "тонкого" клиента на этапе разработки

При работе с данными клиент может сохранять их на своем компьютере, работая в автономном режиме и не загружая сеть передачей информации. Обновленные данные передаются на сервер, а с сервера новые данные загружаются по мере необходимости. Этот принцип работы напоминает работу с кэшированными изменениями и реализуется с помощью компонента `ClientDataSet`.

Для просмотра состояния текущей записи клиентского набора данных используется метод `UpdateStatus: TUpdateStatus`, возвращающий следующие значения:

- `usUnmodified` (в записи нет изменений);
- `usModified` (запись изменена (отредактирована));
- `usInserted` (запись вставлена (является новой));
- `usDeleted` (запись удалена).

Проанализировав состояние текущей записи, можно вывести соответствующее сообщение для пользователя, например, в тексте надписи `Label`. Если код, выполняющий вызов метода и анализ его результата, расположить в обработчике события `DataChange` компонента `DataSource`, который связан с клиентским набором данных, то надпись будет автоматически отображать состояние текущей записи.

Получить доступ ко всем изменениям, сделанным в записях, но еще не отправленным на сервер, позволяют свойства `Data` и `Delta` типа `OleVariant`, первое из которых представляет собой данные клиентского набора данных, а второе — его измененные данные (Delta-данные). Для получения измененных записей в форме располагается еще один компонент `ClientDataSet`, который связывается с первым клиентским набором данных так:

```
ClientDataSet2.Data := ClientDataSet1.Delta;
```

Оба свойства доступны во время выполнения, поэтому их значения можно использовать только программно. После приведенного выше присваивания второй клиентский набор данных будет содержать все неотправленные изменения.

Замечание

Если изменения в записях отсутствуют, то при попытке выполнить присваивание генерируется исключение.

Свойство `ChangeCount` типа `Integer`, доступное во время выполнения, содержит число измененных записей, которое нужно проверять на равенство 0 перед тем, как делать попытку получить эти записи. То есть инструкция присваивания должна иметь следующий вид:

```
if ClientDataSet1.ChangeCount > 0
       then ClientDataSet2.Data := ClientDataSet1.Delta;
```

На рис. 29.6 показана форма клиентского приложения во время его выполнения. В верхней сетке `DBGrid1` отображаются записи клиентского набора данных `ClientDataSet1`, которые доступны для просмотра и изменения. В надписи `Label1`, расположенной над сеткой, выводится состояние текущей записи. Во второй сетке `DBGrid2` отображаются изменения, сделанные в записях клиентского набора данных. Эта сетка через источник данных `DataSource2` связана со вторым клиентским набором данных `ClientDataSet2`. Надписи `Label2` и `Label3` отображают текст `'Изменения в записях — '` и число измененных записей соответственно.

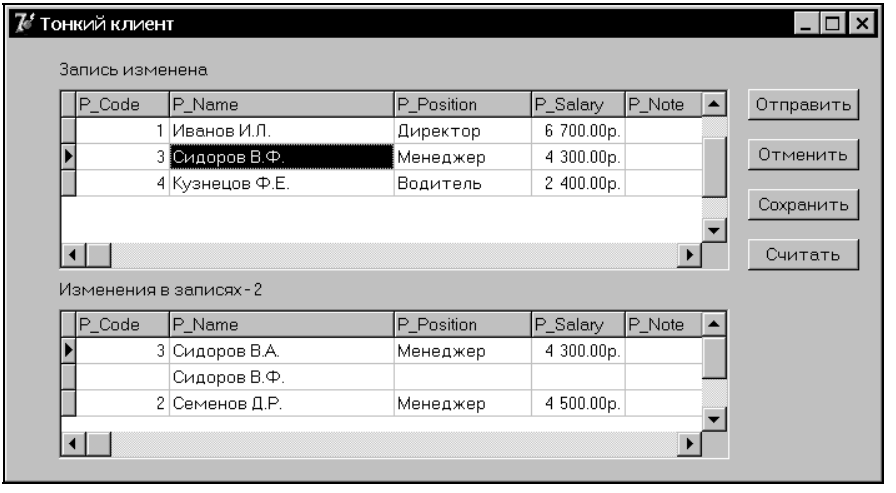


Рис. 29.6. Просмотр состояния текущей записи и изменений в записях

Соответствующий код содержится в обработчике события `DataChange` источника данных `DataSource1`, который связан с клиентским набором данных `ClientDataSet1`:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  case ClientDataSet1.UpdateStatus of
    usUnModified: Label1.Caption := 'Запись не изменялась';
    usModified:   Label1.Caption := 'Запись изменена';
    usInserted:   Label1.Caption := 'Запись вставлена';
    usDeleted:    Label1.Caption := 'Запись удалена';
  end;
  if ClientDataSet1.ChangeCount > 0
       then ClientDataSet2.Data:=ClientDataSet1.Delta;
  Label2.Caption:='Изменения в записях — ' +
             IntToStr(ClientDataSet1.ChangeCount);
end;
```

В форме расположены также четыре кнопки:

- **Отправить** (`Button1`);
- **Отменить** (`Button2`);
- **Сохранить** (`Button3`);
- **Считать** (`Button4`).

Назначение кнопок и обработчики событий их нажатия будут рассмотрены далее.

Сделанные изменения действуют только в приложении клиента и при завершении его работы теряются. Чтобы выполнить обновление данных, изменения нужно отправить на сервер приложений, для чего предназначается метод `ApplyUpdates(MaxErrors: Integer): Integer`. Параметр `MaxErrors` определяет максимальное число ошибок, допустимых при выполнении метода; если для параметра указать значение –1, то на сервер приложений будут переданы все изменения. В качестве результата функция `ApplyUpdates` возвращает число ошибок. Ошибки передачи изменений чаще всего вызваны конфликтами, связанными с редактированием этих же записей другими клиентами.

Рассмотрим в качестве примера следующую процедуру:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label4.Caption := IntToStr(ClientDataSet1.ApplyUpdates(-1));
end;
```

При нажатии кнопки `Button1` на сервер приложений пересылаются все изменения, сделанные в приложении клиента. Число ошибок, связанных с пересылкой записей, выводится в надписи `Label4`.

Пересылка записей и связанное с этим обновление БД может привести к конфликту записей. В этом случае в клиентском наборе данных для каждого такого конфликта генерируется событие `OnReconcileError` типа `TReconcileErrorEvent`. Тип события описан так:

```
type TReconcileErrorEvent = procedure(DataSet: TClientDataSet;
       E: EReconcileError; UpdateKind: TUpdateKind;
       var Action: TReconcileAction) of object;
```

Параметр `DataSet` определяет клиентский набор данных, обновление чьих записей привело к конфликту и вызвало исключение, объект которого содержит параметр `E`. Этот параметр указывает тип операции обновления данных:

- `usModified` (редактирование);
- `usInserted` (вставка);
- `usDeleted` (удаление).

Параметр `Action` определяет действие, которое должно быть выполнено сервером приложений для устранения ошибки:

- `raSkip` — запись остается в приложении клиента без изменений, т. е. остается в Delta-данных (по умолчанию);

- `raAbort` — операция обновления данных прекращается;
- `raMerge` — изменения этого клиента объединяются с изменениями других клиентов; принимаются изменения значений только тех полей, которые не были изменены другими клиентами;
- `raCorrect` — данные, имеющиеся на сервере, заменяются данными этого клиента; данные, поступившие от других клиентов, будут изменены;
- `raCancel` — операция обновления данных прекращается, все изменения удаляются и восстанавливаются первоначальные значения записей;
- `raRefresh` — изменения, сделанные в приложении клиента, сбрасываются и заменяются значениями, имеющимися на сервере.

Для упрощения работы с возникающими ошибками обновления данных удобно использовать стандартный диалог, который добавляется к проекту выбором объекта **Reconcile Error Dialog** (Диалог устранения ошибок) на вкладке **Dialogs** Хранилища объектов.

Для пользователя отображаются: информация о типе операции обновления данных, описание возникшей ошибки, данные конфликтной записи. Составом отображаемых полей управляют два флажка:

- **Show conflicting fields only** (Показывать только конфликтные поля);
- **Show changed fields only** (Показывать только измененные поля).

Пользователь определяет направленное на устранение ошибки действие с помощью группы переключателей **Reconcile Action**:

- **Skip** (Пропуск);
- **Cancel** (Отмена);
- **Correct** (Подтверждение);
- **Refresh** (Обновление);
- **Merge** (Объединение).

Выбор переключателя приводит к установке соответствующего значения параметра `Action` обработчика события `OnReconcileError`.

Рассмотрим следующий пример:

```
// Подключение модуля формы диалога Reconcile Error Dialog
uses reUnit;
...
procedure TForm1.ClientDataSet1ReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

При возникновении ошибки обновления данных вызывается обработчик события `OnReconcileError`. Для предоставления пользователю информации о возникшей ошибке и обеспечения возможности реагировать на нее в теле обработчика вызыва-

ется функция HandleReconcileError. Возвращаемый функцией результат зависит от действий пользователя и присваивается параметру Action обработчика события OnReconcileError. Чтобы можно было реализовать вызов функции HandleReconcileError, ее модуль должен быть включен в список uses.

При возникновении конфликта, связанного с обновлением записи, диалоговое окно устранения ошибки имеет вид, показанный на рис. 29.7.

При отладке приложения на локальном компьютере конфликт обновления можно вызвать, запустив две копии приложения клиента и выполнив попытку редактирования одной и той же записи.

При изменении записей клиентского набора данных можно отменить изменения без передачи их на сервер приложений.

Метод CancelUpdates отменяет все изменения, не отправленные на сервер. Так, выполнение процедуры-обработчика

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  ClientDataSet1.CancelUpdates;
end;
```

приводит к тому, что при нажатии кнопки Button2 все изменения, сделанные в приложении клиента, отменяются.

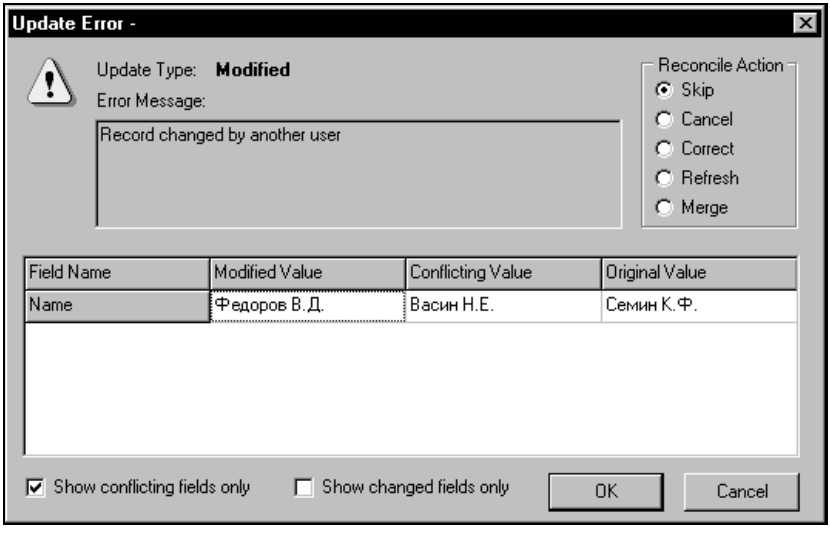


Рис. 29.7. Диалоговое окно устранения ошибок обновления данных

Метод UndoLastChange(FollowChange: Boolean): Boolean отменяет последнее изменение независимо от его вида (редактирование, вставка или удаление записи). Параметр FollowChange управляет позиционированием указателя текущей записи после выполнения отмены. При значении True происходит позиционирование указателя текущей записи на восстановленную запись, а при значении False положение этого указателя не изменяется. В качестве результата функция UndoLastChange возвращает признак успеш-

ности выполнения операции отмены изменения: `True` — операция завершилась успешно, `False` — операция не выполнена.

При организации автономной работы клиента удобно использовать методы `LoadFromFile` и `SaveToFile`.

Метод `LoadFromFile(const FileName: string = '')` загружает в клиентский набор данных данные, ранее сохраненные в файле, имя которого указано параметром `FileName`. Если значением параметра является пустая строка, то данные читаются из файла, задаваемого свойством `FileName`.

Метод `SaveToFile(const FileName: string = ''; Format TDataPacketFormat = dfBinary)` сохраняет данные в файле с именем `FileName`. Необязательный параметр `Format` указывает формат файла:

- `dfBinary` (двоичный файл) — по умолчанию;
- `dfXML` (XML, escape-последовательность);
- `dfXMLUTF8` (XML, последовательность UTF8).

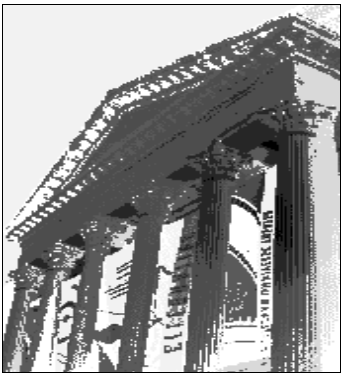
Если на диске нужно сохранить все записи набора данных, то с сервера должны быть получены все данные. С этой целью свойство `PacketRecords` клиентского набора данных следует установить в значение `-1`.

Рассмотрим в качестве примера обмен данными с диском:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
       ClientDataSet1.SaveToFile(SaveDialog1.FileName);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
       ClientDataSet1.LoadFromFile(OpenDialog1.FileName);
end;
```

При нажатии кнопки `Button3` открывается диалог `SaveDialog1` выбора файла для сохранения данных. Выбор файла и нажатие кнопки **Сохранить** приводят к записи данных на диск в указанном файле. По умолчанию файл имеет двоичный формат. Нажатие кнопки `Button4` вызывает диалог `OpenDialog1` выбора файла для чтения данных. Выбор файла и нажатие кнопки **Открыть** приводит к загрузке в клиентский набор ранее сохраненных данных.

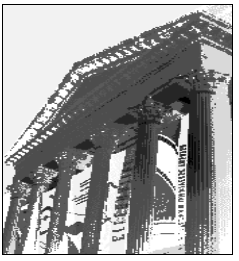
Для указания имени файла можно использовать свойство `FileName` типа `String`. Если это свойство задано, то методы `LoadFromFile` и `SaveToFile` вызываются без параметров.



ЧАСТЬ VI

Базы данных и Интернет

ГЛАВА 30



Введение в технологии публикации баз данных в Интернете

Публикация баз данных в Интернете — это размещение информации из баз данных на Web-страницах сети. Отметим, что такая публикация связана с решением следующих типичных задач, встающих перед разработчиками современного программного обеспечения:

- организация взаимосвязи СУБД, работающих на различных платформах;
- построение информационных систем в Интернете на основе многоуровневой архитектуры БД (архитектура таких систем включает дополнительный уровень — Web-сервер с модулями расширения серверной части, который и реализует возможность информационного обмена и публикации БД в глобальной сети);
- построение локальных интранет-сетей на основе технологии публикации БД в Интернете (локальные сети строятся на принципах Интернета с выходом при необходимости в глобальную сеть);
- использование в Интернете информации из существующих локальных сетевых баз данных (при необходимости опубликования в глобальной сети информации из локальных сетей);
- применение БД для упорядочивания (каталогизирования) информации (огромный объем данных, представленных в Интернете, не обладает требуемой степенью структурированности, что делает процесс поиска необходимой информации весьма сложным и долгим);
- применение языка SQL для поиска необходимой информации в БД;
- использование средств СУБД для обеспечения безопасности данных, разграничения доступа и управления транзакциями при создании интернет-магазинов, защищенных информационных систем и т. д.;
- стандартизация пользовательского интерфейса на основе применения Web-обозревателей с типовым внешним видом пользовательского интерфейса и его типовой реакцией на действия пользователя;
- использование Web-обозревателя в качестве дешевой клиентской программы для доступа к БД.

Размещение информации из БД в Интернете представляет собой новую информационную технологию, получившую широкое распространение в последнее время в связи с ростом популярности и доступности "Всемирной паутины". В данной главе мы рассмотрим базовые элементы интернет-технологий, являющиеся основой для разработки Web-приложений.

В Интернете вся информация размещается на Web-страницах, для написания которых используются язык HTML (HyperText Markup Language — язык разметки гипертекста) или его расширения, такие как DHTML (Dynamic HTML — динамический HTML) и XML (eXtensible Markup Language — расширяемый язык разметки). В состав Web-страницы могут входить текстовая информация, ссылки на другие Web-страницы, графические изображения, аудио- и видеоинформация и другие данные. Web-страницы хранятся на Web-сервере.

Для доступа к Web-страницам используются специальные клиентские программы — Web-обозреватели (программы просмотра, называемые также Web-браузерами — от англ. *browser*), находящиеся на компьютерах пользователей. Обозреватель формирует запрос на получение требуемой страницы или другого ресурса с помощью специального адреса URL (Uniform Resource Locator — универсальный указатель ресурса). Этот адрес определяет тип протокола для передачи ресурса, имя домена, используемое для доступа к Web-узлу, номер порта, локальный путь к файлу и дополнительные аргументы. Соединение с Web-узлом устанавливается с помощью протокола передачи данных HTTP (HyperText Transfer Protocol — протокол передачи гипертекста).

Для расширения возможностей клиента (обозревателя) и Web-сервера создаются соответствующие программы расширения. Схема взаимодействия клиента и Web-сервера с использованием программ расширения приведена на рис. 30.1.

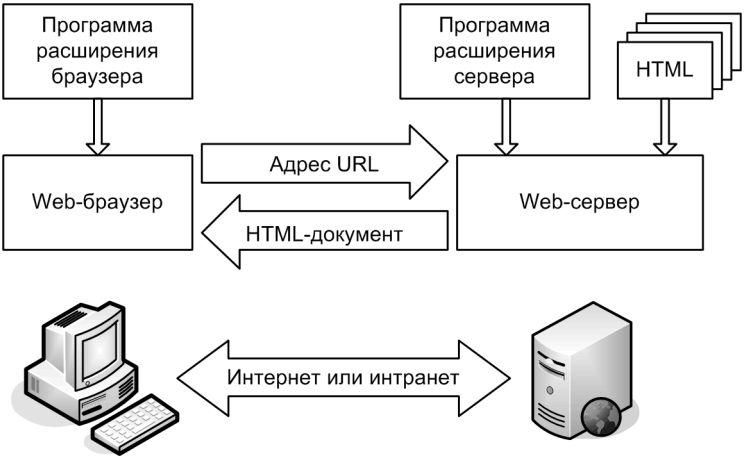


Рис. 30.1. Схема взаимодействия обозревателя и Web-сервера

При организации такого взаимодействия могут использоваться следующие средства:

- сценарии (скрипты), подготавливаемые на различных языках сценариев (JavaScript, JScript и VBScript) и вставляемые в Web-документ;

- апплеты и сервлеты на языке Java;
- элементы управления ActiveX;
- консольные исполняемые программы, реализованные с использованием интерфейса CGI;
- исполняемые программы, реализованные с использованием интерфейса WinCGI;
- динамические библиотеки, реализованные с использованием интерфейса ISAPI;
- динамические страницы IDC/HTX;
- активные серверные страницы ASP.

Основные элементы интернет-технологий

В этом подразделе мы дадим краткую характеристику перечисленным выше средствам, которые хотя и выходят за рамки Delphi, но составляют основные элементы интернет-технологий и используются при публикации баз данных.

Сценарии JavaScript, JScript и VBScript

Сценарии (скрипты), написанные на языках JavaScript, JScript или VBScript, используются для динамического управления интерфейсными объектами (компонентами) Web-документа. Эти языки являются интерпретируемыми, т. е. исполняемый файл не создается, а код выполняется непосредственно в процессе интерпретации. Интерпретацию и выполнение сценариев осуществляет обозреватель или Web-сервер. Скрипты рассматриваются как расширение языка HTML и могут включаться в тело HTML-документа. Часть сценария может исполняться во время загрузки документа, а часть сценария, реализованная, как правило, в виде функции, выполняться в ответ на действия пользователя. Использование того или иного языка сценариев определяется типом применяемого обозревателя.

Язык JavaScript представляет собой объектно-ориентированный язык, имеющий С-подобный синтаксис. Сценарии на JavaScript используются в обозревателе Netscape Navigator, а сценарии на языке JScript — в Microsoft Internet Explorer. Лексика и синтаксис этих языков практически идентичны, различия заключаются в объектных моделях, используемых обозревателями (т. е. в совокупностях элементов, их атрибутов и событий Web-документа). Internet Explorer дополнительно поддерживает язык VBScript, функционально эквивалентный языку JScript, но более простой в освоении. Этот язык наиболее привычен для программистов, имеющих опыт работы с Microsoft Visual Basic.

Сценарии могут применяться как расширение обозревателя (клиентское расширение) или как расширение Web-сервера. В первом случае сценарии включаются в Web-документ и используются для создания динамических эффектов при просмотре Web-страницы, а во втором они требуются для динамического создания Web-документов в ответ на запрос пользователя. Для уменьшения нагрузки на Web-сервер часть функций, связанных с предварительной обработкой запросов и введенных данных, целесообразно выполнять на стороне клиента (в Web-обозревателе). В этом случае говорят об

активности на стороне клиента. Кроме сценариев, такая активность может быть реализована и другими средствами (апплеты Java, элементы управления ActiveX).

Элементы управления ActiveX

Элементы управления ActiveX представляют собой модули расширения, которые могут использоваться и на стороне клиента, и на стороне сервера. Они реализуются с помощью динамических библиотек DLL и встраиваются в Web-документ как дополнительные интерфейсные элементы. Механизм работы элементов управления ActiveX позволяет получать с их помощью неограниченный доступ к локальным ресурсам компьютера пользователя, включая возможность передачи на сервер любой информации с этого компьютера. Поэтому использование элементов ActiveX на стороне клиента в Интернете не всегда оправданно с точки зрения обеспечения безопасности данных.

Код объектов ActiveX может содержать вирус. Если загрузить такой объект, то компьютеру клиента в принципе может быть причинен вред. Для частичного устранения этого недостатка компания Microsoft добавляет в свои программы цифровую подпись (signing), которая гарантирует "чистоту" объектного кода. Кроме того, обозреватель Internet Explorer способен идентифицировать "подозрительные" модули и выдает при необходимости предупреждающее сообщение, предоставляя пользователю возможность отменить выполнение кода. Однако этот механизм не устраняет целиком опасность заражения вирусом компьютеров пользователей, использующих объекты ActiveX.

Если в вызванной Web-странице встречаются новые элементы управления ActiveX, то их предварительно требуется загрузить и установить, а это нарушает универсальность обозревателя. Поэтому элементы управления ActiveX рекомендуется использовать преимущественно на стороне сервера.

Апплеты и сервлеты Java

Апплеты Java (applet — "маленькое приложение") применяются для создания динамически формируемого интерфейса пользователя. Язык Java является объектно-ориентированным языком с синтаксисом, похожим на синтаксис языка C++. Однако возможности языка Java по доступу к локальным ресурсам пользователей сильно урезаны, что делает его безопасным для использования в сети. Апплеты предназначены для выполнения на любых платформах. Их код интерпретируется виртуальной Java-машиной, входящей в состав обозревателя. Использование такого механизма гарантирует целостность локальных данных пользователя. Для включения апплета в Web-страницу применяется специальный тег (tag — метка) языка HTML. *Сервлеты*, в отличие от апплетов, выполняются на стороне сервера и служат для обработки запросов, передаваемых от обозревателя.

Интерфейсы CGI и WinCGI

Для создания модулей расширения Web-сервера могут использоваться *интерфейсы CGI* (Common Gateway Interface — общий шлюзовой интерфейс) и *API* (Application Program Interface — интерфейс прикладного программирования).

Интерфейс CGI является стандартным протоколом взаимодействия между Web-сервером и модулями расширения, которые могут применяться для выполнения дополнительных функций, не поддерживаемых сервером. Например, такие модули используются для обработки получаемой от пользователя информации, для динамического формирования Web-документа, публикации БД на Web-странице и т. д.

Интерфейсу CGI соответствуют обычные консольные приложения операционной системы DOS. Обмен информацией между сервером и модулем расширения осуществляется с помощью стандартного потокового ввода/вывода, а передача управляющих параметров организуется через переменные окружения операционной системы или через параметры URL-адреса модуля расширения.

Для запуска модуля расширения достаточно задать его URL-адрес в строке обозревателя и начать загрузку документа. При получении запроса обозревателя к CGI-приложению сервер запускает это приложение и передает ему данные из командной строки запроса. CGI-приложение формирует ответ и помещает его в выходной поток (на стандартном устройстве вывода), затем сервер посылает этот ответ с использованием протокола HTTP обратно обозревателю. В случае параллельной обработки нескольких запросов сервер запускает отдельный процесс для обработки каждого запроса, причем для каждого процесса создается копия модуля расширения в памяти компьютера, на котором находится Web-сервер. Поэтому недостатками протокола CGI является невысокая скорость обработки запросов и повышенная загрузка Web-сервера.

Существует адаптированный вариант общего CGI-протокола для среды Windows 3.1 — WinCGI. Он отличается от протокола CGI тем, что управляющие параметры передаются через INI-файл, а входной и выходной потоки данных перенаправлены в специальные файлы. В остальном механизм взаимодействия с сервером аналогичен механизму, используемому протоколом CGI.

Интерфейсы ISAPI/NSAPI

Более перспективными интерфейсами являются *интерфейсы ISAPI/NSAPI* (Internet Server API/Netscape Server API), разработанные фирмами Microsoft и Netscape соответственно. Эти интерфейсы также предназначены для разработки дополнительных модулей расширения Web-сервера. В случае использования этих интерфейсов модули расширения реализуются в виде библиотек DLL. Запуск модуля расширения выполняется сервером в ответ на запрос обозревателя на загрузку URL-адреса этого модуля. Взаимодействие между сервером и модулем расширения осуществляется с помощью специальных объектов (Request, Response). Сервер передает параметры запроса модулю расширения и получает сформированный Web-документ, который с помощью протокола HTTP пересылается обратно обозревателю.

При многопользовательском режиме работы сервера загрузка ISAPI-модуля расширения (библиотеки DLL) происходит один раз при первом обращении. При обработке последующих запросов к модулю расширения сервер использует уже загруженный экземпляр динамической библиотеки. Такой механизм взаимодействия сервера и модуля расширения обеспечивает экономию ресурсов сервера и увеличение скорости обработки запросов. Однако при возникновении ошибок в коде модуля расширения сам Web-

сервер аварийно завершит работу (в отличие от интерфейса CGI, при использовании которого в случае возникновения ошибки в модуле расширения сервер может продолжать нормально функционировать). Поэтому для отладки модулей расширения, реализуемых с помощью интерфейса ISAPI, создают аналог модуля на базе интерфейса CGI. После успешной отладки этого аналога модуль расширения переводится на базу ISAPI.

Страницы ASP, PHP и IDC/HTX

Страницы ASP, PHP и IDC/HTX — это специальные типы страниц, используемые для динамического формирования на сервере Web-документов, содержащих информацию из БД.

Страница IDC (Internet Database Connector) содержит псевдоним БД (системную запись, используемую операционной системой для связи с базой данных), SQL-запрос к базе данных, идентификатор пользователя и пароль для доступа к БД.

Страница HTX (страница гипертекста) содержит HTML-шаблон, определяющий, какую информацию и в каком формате будет иметь результирующий файл. Этот файл поддерживает все теги языка HTML и дополнительные теги для размещения информации из БД.

Страница ASP (Active Server Pages — активные серверные страницы) содержит одновременно HTML-шаблон и SQL-запрос к БД. В ASP-странице используются средства языка JScript и объектная модель ASP, с помощью которых организуется доступ к БД и формируется внешний вид создаваемой Web-страницы. В ASP-страницах так же, как в IDC-страницах, поддерживаются все теги языка HTML и используются дополнительные теги для размещения кода на языке JScript.

Страницы PHP (Personal Home Page — персональная домашняя страница) разрабатываются с помощью одноименного языка обработки сценариев, команды которого включаются в документ HTML.

ASP-, PHP- и IDC/HTX-страницы обрабатываются Web-сервером, в результате чего генерируется Web-страница, содержащая информацию из БД, которая отсылается обозревателю.

Формирование Web-страниц

В Интернете информация находится на Web-узлах (Web-сайтах), на которых для организации взаимодействия с пользователями сети устанавливается специальное программное обеспечение, в том числе Web-сервер. В функции Web-сервера входит обработка запросов Web-обозревателей пользователей сети. В результате обработки запроса сервер формирует Web-документ, который отсылается обозревателю в формате протокола HTTP.

Web-сервер может формировать динамические Web-страницы и отсылать готовые Web-страницы различными способами. Для формирования динамической Web-страницы, содержащей информацию из БД, дополнительно используются модули расширения сервера (см. рис. 30.1).

Различают пассивное и активное состояния Web-сервера. Так, Web-сервер находится в *пассивном* состоянии, если формируемый им документ содержит статическую текстовую, графическую, мультимедийную информацию и гиперссылки. В таком документе отсутствуют средства ввода и обработки запросов к серверу.

В случае, когда на Web-странице находятся интерфейсные элементы, которые могут в ответ на реакцию пользователя обращаться с запросами к серверу, сервер переходит в *активное* состояние. Для публикации БД основной интерес представляет активный Web-сервер, реализуемый с помощью программных расширений. Для создания программных расширений Web-сервера, формирующих на Web-странице содержимое БД, используются следующие средства:

- консольные исполняемые программы, использующие интерфейс CGI;
- исполняемые программы, использующие интерфейс WinCGI;
- динамические библиотеки, использующие интерфейс ISAPI;
- динамические страницы IDC/HTX;
- активные серверные страницы ASP.

Кроме того, для организации связи программных расширений Web-сервера с БД используются современные интерфейсы доступа к данным, такие как OLE DB, ADO и ODBC. Эти интерфейсы представляют промежуточный уровень между источником данных и приложением, в качестве которого здесь выступают программные расширения Web-сервера.

Интерфейсы OLE DB, ADO, ODBC

Интерфейсы доступа к источникам данных OLE DB, ADO, используемые в продуктах Microsoft, существенно отличаются от предшествующих интерфейсов, подобных ODBC.

Интерфейс ODBC (Open Database Connectivity — совместимость открытых баз данных) применяется операционной системой для доступа к источникам данных, как правило, к реляционным БД, использующим язык структурированных запросов SQL для организации управления данными.

Интерфейс OLE DB (Object Linking and Embedding DataBase — связывание и внедрение объектов баз данных) является более универсальной технологией для доступа к любым источникам данных через стандартный интерфейс COM (Component Object Model — объектная модель компонентов). Данные могут быть представлены в любом виде и формате (например, реляционные БД, электронные таблицы, документы в RTF-формате и т. д.). В интерфейсе OLE DB используется механизм провайдеров, под которыми понимаются поставщики данных, составляющих надстройку над физическим форматом данных. Такие провайдеры еще называются сервис-провайдерами, благодаря им можно объединять в однотипную совокупность объекты, связанные с разными источниками данных.

Кроме того, есть OLE DB-провайдер, который реализует интерфейс доступа OLE DB поверх конкретного сервис-провайдера данных. При этом поддерживается возмож-

ность многоуровневой системы провайдеров OLE DB, когда провайдер OLE DB может находиться поверх группы провайдеров OLE DB или сервис-провайдеров.

Интерфейс OLE DB может использовать для доступа к источникам данных интерфейс ODBC. В этом случае применяется провайдер OLE DB для доступа к ODBC-данным. Таким образом, интерфейс OLE DB не заменяет интерфейс ODBC, а позволяет организовывать доступ к источникам данных через различные интерфейсы, в том числе и через ODBC.

Интерфейс ADO (ActiveX Data Objects — объекты данных ActiveX) предоставляет иерархическую модель объектов для доступа к различным OLE DB-провайдерам данных. Он характеризуется еще более высоким уровнем абстракции и базируется на интерфейсе OLE DB. Объектная модель ADO включает небольшое количество объектов, которые обеспечивают соединение с провайдером данных, создание SQL-запроса к данным, создание набора записей на основе запроса и др. Разрабатывая интерфейс ADO, фирма Microsoft предназначала его для использования в сетях Интернет/интранет для доступа к различным источникам данных.

Статическая публикация БД

При публикации БД на Web-страницах используются следующие способы формирования Web-страниц:

- статическая публикация Web-страниц, содержащих информацию из БД;
- динамическая публикация Web-страниц, содержащих информацию из БД.

Рассмотрим особенности формирования каждого типа страниц.

В случае *статической* публикации БД Web-страницы с соответствующей информацией создаются и хранятся на Web-сервере до поступления запроса пользователя на их получение (в виде файлов на жестком диске в формате Web-документа). Генерацию таких страниц может выполнять обычное приложение Windows, имеющее доступ к БД. Этот способ используется при публикации информации, редко обновляемой в базе данных (обновление БД можно выполнять с требуемой периодичностью или при внесении в нее изменений). Такая организация публикации БД в Интернете имеет ряд преимуществ, заключающихся в получении более быстрого доступа к Web-документам и уменьшении нагрузки на сервер при обработке запросов.

Отметим, что при обработке запроса на получение Web-страницы статическим способом сервер может находиться как в пассивном, так и в активном состоянии. В активном состоянии сервер находится в случае, если Web-страницы содержат интерактивные элементы, которые в ответ на реакцию пользователя обращаются с запросами к серверу.

Динамическая публикация БД

Динамическая публикация используется в случаях, когда необходимо публиковать информацию из БД, содержимое которой часто обновляется. Таким способом публикуется информация из БД для интернет-магазинов и информационных систем, работающих в реальном масштабе времени, например, систем продажи билетов.

При динамической публикации страницы создаются после поступления запроса пользователя на сервер. Сервер передает запрос на генерацию таких страниц своей программе-расширению, которая формирует требуемый документ. Затем сервер отсылает готовые Web-страницы обратно обозревателю. Для формирования динамических страниц используются различные средства и технологии: ASP и IDC/HTX-страницы, программы расширения сервера на основе интерфейсов CGI и ISAPI.

В случае использования ASP- и IDC/HTX-страниц запрос на получение динамически формируемой Web-страницы передается специальным динамическим библиотекам, входящим в состав Web-сервера. Например, если используется Personal Web Server и публикация осуществляется средствами IDC/HTX, то применяется динамическая библиотека "httpodbc.dll". Такие библиотеки анализируют файл ASP или файлы IDC и HTX, которые используются в качестве шаблона.

Путь к файлу ASP или IDC задается в строке запроса. В зависимости от расширения имени файла в строке запроса сервер принимает решение о передаче управления тому или иному модулю расширения: если указывается расширение exe, то используется интерфейс CGI, а если dll — то интерфейс ISAPI.

Отметим, что при формировании динамической Web-страницы сервер находится в активном состоянии. После отправки страницы обозревателю сервер может перейти в пассивное состояние, если сформированная страница содержит только статическую информацию. На одном Web-сервере могут использоваться страницы, создаваемые как статическим, так и динамическим способом.

Web-приложения

Среди программных средств, используемых для получения информации из Интернета, выделилась новая категория программ — *Web-приложения*, или интернет-приложения. Web-приложение — это совокупность Web-страниц, клиентских и серверных сценариев, расположенных на одном или нескольких компьютерах и выполняемых в рамках одной информационной системы (целевой задачи). Для разработки Web-приложений могут использоваться различные комбинации рассмотренных технологий.

Информационные системы, построенные на основе Web-приложений, характеризуются многоуровневой архитектурой и позволяют использовать все достоинства Интернета. Приложения, реализующие технологию публикации БД в Интернете, составляют отдельный класс Web-приложений. В принципе такие программы представляют собой многоуровневые клиент-серверные приложения. Однако Web-приложения имеют свои особенности, связанные с организацией работы Интернета. Разработчику Web-приложения приходится учитывать следующие аспекты:

- совместимость Web-обозревателей;
- разграничение доступа и обеспечение безопасности данных;
- надежность линий связи.

Web-приложения, публикующие БД на Web-страницах, выполняются на стороне сервера. Сервер обрабатывает запросы обозревателя. Запросы к БД Web-сервер передает

серверу приложений или серверу баз данных. Обработав запрос, сервер БД передает нужные данные Web-серверу, который формирует Web-документ и отсылает его обозревателю.

Функции обозревателя заключаются в отображении Web-страниц, сгенерированных сервером или модулями расширения, и отправке запросов пользователя Web-приложению, т. е. обозреватель является связующим звеном между пользователем и Web-приложением.

Протоколы передачи данных

В Интернете могут использоваться различные платформы и Web-серверы: для операционных систем Windows NT/2000 — это Microsoft Internet Information Server (MS IIS), для UNIX — Apache Server (этот сервер распространяется в исходных кодах, поэтому после перекомпиляции может выполняться и в других операционных системах), а для Windows 98 — Microsoft Personal Web Server.

Запросы от обозревателя поступают на Web-сервер в соответствии с установленным протоколом обмена, например, HTTP. Вообще, в Интернете поддерживается более десятка протоколов. Протокол HTTP является одним из самых распространенных и предназначен для передачи данных различных форматов между обозревателем и Web-сервером. Соединение между компьютером-отправителем и компьютером-получателем осуществляется с помощью протокола низкого уровня TCP/IP (Transport Control Protocol/Internet Protocol — протокол управления передачей/межсетевой протокол).

Протокол TCP/IP представляет собой универсальный платформо-независимый протокол передачи данных по сети. За физическую передачу данных отвечает протокол IP. TCP представляет собой протокол более высокого уровня. Он разбивает в компьютере-отправителе файлы на пакеты, добавляет в каждый из них адрес получателя и порядковый номер пакета в группе пакетов. В компьютере-получателе протокол TCP собирает файлы из пакетов, проверяет их целостность и при необходимости посылает запрос на повторную трансляцию.

При отправке серверу HTTP-запроса на формирование динамической страницы, содержащей отчет из БД, сервер включает в работу модули расширения. Взаимодействие Web-сервера и модуля расширения основывается на интерфейсах CGI или API. Модуль расширения получает запрос и обрабатывает его, формируя ответ (обычно Web-страницу).

Универсальный указатель ресурсов (URL)

Обозреватель формирует запрос на получение нужной страницы с помощью универсального указателя ресурса — URL (Universal Resource Locator). Для загрузки требуемой страницы в окне обозревателя указывается строка адреса. Загрузку требуемого документа можно осуществить и из Web-документа с помощью специальных тегов.

Приведем структуру URL для протокола HTTP:

```
http://<хост>/<порт>:<путь>?<поиск>
```

Здесь:

- `http` — имя протокола, используемого для доступа к ресурсу;
- `<хост>` — имя домена, используемое для поиска требуемого Web-узла в Интернете;
- `<порт>` — номер порта, который задает номер логического канала связи в Интернете;
- `<путь>` — локальный путь к файлу;
- `<поиск>` — дополнительные параметры запроса.

В обозревателях используются следующие два типа адресов указателей.

- *Абсолютные указатели* задают адрес, полностью определяющий компьютер, каталог и файл. В отличие от относительных, абсолютные адреса могут ссылаться на файлы, расположенные на других компьютерах.
- *Относительные указатели* используются для получения доступа к файлу, расположенному на том же компьютере, что и документ, в котором находится указатель на этот файл. Например, если обозреватель загрузил страницу, находящуюся по адресу **http://myweb**, то при задании относительного адреса **/my** реальный адрес будет **http://myweb/my**. Относительные адреса удобны для использования в пределах одного компьютера.

Основу Web-приложений составляет совокупность Web-документов и различных шаблонов, использующих формат языка HTML или его расширений. В связи с этим для разработки Web-приложений прежде всего необходимо знать HTML. Далее рассматриваются конструкции языков HTML и XML, используемые для разработки Web-приложений, публикующих БД в Интернете.

Использование HTML

В этом разделе рассматриваются состав HTML-документа и назначение его тегов.

Состав HTML-документа

Язык HTML представляет собой универсальный язык подготовки документов для Интернета. HTML-документ является разновидностью Web-документа и в общем случае представляет собой обычный ASCII-файл, который можно отредактировать в любом текстовом редакторе. Он состоит из текстовых данных, снабженных специальными управляющими метками — тегами. Теги несут в себе служебную информацию для обозревателя, задавая различные режимы форматирования.

Теги заключаются в угловые скобки. Например, начальным тегом документа является следующий тег:

```
<HTML>
```

Теги могут быть парными и непарными. Парные теги делятся на открывающий и закрывающий. Пара таких тегов называется *контейнером*.

Например, весь текст HTML-документа заключен в теги <HTML> и </HTML>:

```
<HTML>
...
<!-- Это комментарий, который в обозревателе не отображается.
     Между тегами <HTML> и </HTML> располагается
     текст HTML-документа с другими управляющими тегами --!>
...
</HTML>
```

Здесь <HTML> — открывающий тег, </HTML> — закрывающий тег, между ними находятся все данные. <!-- Комментарий --!> — пример непарного тега.

HTML-документ состоит из двух частей — заголовка и тела документа. *Заголовок* является необязательным элементом, он может использоваться для того, чтобы задать название документа, определить отношения между несколькими документами и т. д. Заголовок и тело документа обрамляются специальными тегами. В *теле* HTML-документа находятся текст документа и различные управляющие теги.

Рассмотрим подробнее основные группы тегов, которые могут понадобиться для создания собственного Web-приложения, и приведем примеры их практического использования.

Структурные теги

Структурные теги отвечают за выделение основных разделов (частей) HTML-документа. К ним относятся теги границ документа (<HTML> и </HTML>), теги заголовка документа (<HEAD> и </HEAD>), теги тела документа (<BODY> и </BODY>).

Тег <HTML> контейнерного типа указывает начало HTML-документа, а тег </HTML> ставится в последней строке документа. Эти теги сообщают обозревателю, что находящиеся между ними строки представляют собой единый HTML-документ, а не обыкновенный файл в формате ASCII.

Замечание

Хотя большинство современных обозревателей могут опознать и не содержащий тегов <HTML> и </HTML> документ, желательно употреблять эти теги, учитывая разнообразные области применения HTML (электронная почта и др.).

Внутри тегов <HEAD> и </HEAD>, определяющих заголовок документа, могут находиться другие теги, задающие название документа и иную общую информацию. Например, название документа, которое обычно показывается в заголовке окна обозревателя, записывают между тегами <TITLE> и </TITLE> в виде текстовой строки.

Тело документа задается с помощью тегов <BODY> и </BODY> в следующем формате:

```
<BODY атрибуты>
    Содержимое документа
</BODY>
```

Здесь атрибуты не обязательны и определяют внешний вид документа в обозревателе; Содержимое документа представляет собой совокупность любых допустимых элементов HTML-документа.

Например, простейший HTML-документ, содержащий заголовок и одну строку текста, имеет следующий вид:

```
<HTML>
<HEAD> <!-- Это заголовок документа --!>
  <TITLE> Это заголовок простого HTML-документа </TITLE>
</HEAD>
<BODY> <!-- Это тело документа --!>
  Это текст простого HTML-документа!
</BODY>
</HTML>
```

Теги форматирования текста

Теги форматирования текста используются для того, чтобы HTML-документ был более наглядным и читаемым. Приведем некоторые из наиболее часто используемых тегов этой группы.

Разделение текста на абзацы осуществляется с помощью тега контейнерного типа `<P ALIGN=Атрибут>...</P>`. Применение этого тега позволяет разделять текст на абзацы независимо от параметров окна обозревателя. Атрибут `ALIGN` может принимать значения `LEFT`, `CENTER` и `RIGHT`, задающие выравнивание текста по левому краю, по центру и по правому краю соответственно. По умолчанию (если атрибут не указан) выравнивание текста осуществляется по левому краю.

Перевод каретки (каретка указывает на текущую позицию, в которой будет производиться следующий вывод текста) на следующую строку в любом месте HTML-документа осуществляется тегом разрыва строки `<BR>`. В отличие от тега абзаца, тег `<BR>` не пропускает строку.

Для повышения читаемости HTML-документа используются заголовки. Тег `<H1>` служит для задания заголовка, является контейнерным и имеет открывающий (`<H1 ALIGN=Атрибут>`) и закрывающий (`</H1>`) теги. HTML располагает шестью уровнями (задаваемыми относительными размерами шрифта) заголовков: `H1` (самый верхний), `H2`, `H3`, `H4`, `H5` и `H6` (самый нижний). Например, задание заголовка со вторым уровнем размера шрифта имеет вид:

```
<H2> Текст из файла </H2>
```

Выравнивание заголовков, как и в случае выравнивания абзацев, осуществляется с помощью атрибута `ALIGN`.

Тег `<HR>` позволяет провести рельефную горизонтальную линию в окне большинства обозревателей, он не требует закрывающего тега. При указании этого тега до и после линии автоматически вставляется пустая строка.

Тег `<I>` является контейнерным и используется для выделения текста курсивом. Например:

```
<I> Этот текст выделен курсивом </I>
```

Контейнерный тег `<B>` служит для выделения фрагментов текста полужирным шрифтом. Например:

```
<B> Этот текст выделен полужирным шрифтом </B>
```

Контейнерный тег <TT> применяется в случаях, когда нужно вывести текст шрифтом, символы которого имеют фиксированную ширину. Например:

```
<TT> Этот текст выведен с использованием шрифта фиксированной ширины </TT>
```

Подчеркивание текста реализуется с помощью контейнерного тега <U>. Например:

```
<U> Подчеркнутый текст </U>
```

Задание различных атрибутов шрифта производится контейнерным тегом <FONT>. После открывающего тега обязательно указать атрибуты, без которых элемент не влияет на помещенный в контейнер текст. Рассмотрим назначение некоторых атрибутов данного тега.

Атрибут FACE используется для задания гарнитуры шрифта, которым обозреватель отображает соответствующий текст. При отсутствии шрифта нужной гарнитуры обозреватель берет шрифт с гарнитурой, установленной по умолчанию. Например:

```
<FONT FACE="Arial"> Этот текст выведен с использованием шрифта Arial </FONT>
```

Атрибут SIZE служит для указания размера шрифта в условных единицах от 1 до 7. Например:

```
<FONT SIZE=1> Текст выводится с использованием размера шрифта 1 </FONT>
```

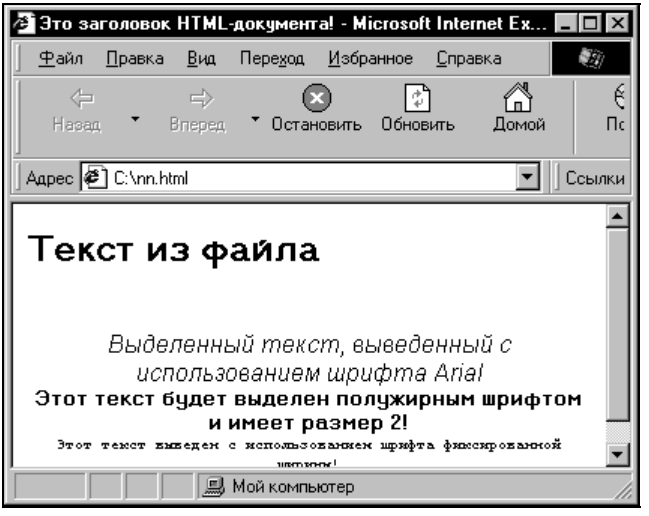


Рис. 30.2. Вид документа с форматированием текста в окне обозревателя

Пример простейшего HTML-документа, в котором использовано форматирование текста:

```
<HTML>
<HEAD> <!-- Это заголовок документа --!>
  <TITLE> Это заголовок HTML-документа </TITLE>
</HEAD>
<BODY> <!-- Это тело документа --!>
  <H2> Текст из файла </H2>
```

```
  <BR>
  <P ALIGN=CENTER>
  <FONT FACE= "Arial">
    <I> Выделенный текст, выведенный с использованием шрифта Arial </I>
  </FONT>
  <BR>
  <FONT SIZE=2>
    <B> Этот текст будет выделен полужирным  шрифтом и имеет размер 2! </B>
    <BR>
    <TT> Этот текст выведен с использованием шрифта фиксированной ширины! </TT>
    </P>
</BODY>
</HTML>
```

Вид HTML-документа, приведенного в этом примере, в окне обозревателя Microsoft Internet Explorer показан на рис. 30.2.

Табличные теги

Для создания таблиц используется табличный тег <TABLE>, являющийся контейнерным. В этом контейнере размещается содержимое таблицы. В теге <TABLE> могут задаваться следующие атрибуты:

- BORDER — определение рамки таблицы; ширина рамки устанавливается в пикселах, например, BORDER=2 (по умолчанию 1);
- ALIGN — выравнивание таблицы в окне обозревателя; может принимать значения LEFT (по умолчанию), CENTER и RIGHT.

Структура таблицы определяется с помощью следующих тегов-контейнеров:

- <TH> — задает заголовки столбцов (строк), которые выделяются полужирным шрифтом и центрируются в своих ячейках;
- <TR> — задает начало строки, а </TR> — конец строки;
- <TD> — задает данные в ячейках.

Приведем пример, в котором задается таблица, содержащая 3 строки и 3 столбца.

```
<TABLE ALIGN=LEFT BORDER=1>
<TR>
  <TH> Заголовок 1-го столбца </TH>
  <TH> Заголовок 2-го столбца </TH>
  <TH> Заголовок 3-го столбца </TH>
</TR>
<TR>
  <TD> Данные 2-й строки и 1-го столбца </TD>
  <TD> Данные 2-й строки и 2-го столбца </TD>
  <TD> Данные 2-й строки и 3-го столбца </TD>
</TR>
<TR>
  <TD> Данные 3-й строки и 1-го столбца </TD>
  <TD> Данные 3-й строки и 2-го столбца </TD>
```

```
  <TD> Данные 3-й строки и 3-го столбца </TD>
</TR>
</TABLE>
```

Аналогично задаются таблицы с произвольным числом строк и столбцов.

Теги определения кадров

Под *кадром* (или фреймом, от англ. *frame*) понимается некоторая отдельная часть окна обозревателя. Каждому кадру соответствует свой HTML-документ, содержащий отображаемую в нем информацию. Кадры определяются внутри определения блока кадров, называемого FRAMESET.

Для определения блока кадров предназначен контейнерный тег <FRAMESET>, имеющий следующий формат

```
<FRAMESET Атрибут1=Значение ... АтрибутN=Значение>
... Теги кадров ...
</FRAMESET>
```

Внутри этого тега-контейнера могут располагаться только теги <FRAME>, определяющие одиночные кадры, или другие контейнеры <FRAMESET>. Рассматриваемый тег замещает тег <BODY>. В HTML-документе, где есть блок кадров FRAMESET, не должно быть тега <BODY>, иначе кадровая структура будет игнорирована.

Рассмотрим применение атрибутов ROWS и COLS тега <FRAMESET>, определяющих строки и столбцы как отдельные кадры в структуре кадров. В HTML-документе можно определить любое число строк и столбцов как кадров. При этом необходимо задать значение для хотя бы одного из атрибутов ROWS или COLS. Тег <FRAMESET> с атрибутами ROWS и COLS имеет следующий формат:

```
<FRAMESET ROWS="Список_значений" COLS="Список_значений"> </FRAMESET>
```

Здесь Список_значений содержит перечисляемые через запятую значения атрибутов, определяющие размеры кадров в пикселах, процентах или относительных единицах.

Следует учитывать, что кадр не может быть единственным. Для того чтобы обозреватель отобразил структуру кадров, необходимо задать в списке значений атрибутов ROWS или COLS больше одного значения. Например,

```
<FRAMESET ROWS="50,50,100,100">
...
<FRAMESET COLS="50%,50%">
```

Здесь первая строка определяет набор кадров из четырех строк, высота которых составляет 50, 50, 100 и 100 пикселов соответственно, а вторая строка определяет набор кадров из двух столбцов.

Высоту строки в пикселах задавать нежелательно, т. к. при этом не учитывается, что окна обозревателей могут иметь различную величину. В абсолютных единицах можно указывать размеры кадра лишь для небольших изображений, в остальных случаях лучше пользоваться относительными величинами, например:

```
<FRAMESET ROWS="25%,50%,25%">
```

Приведенный код создает три строки кадров высотой в 25, 50 и 25% от высоты развернутого окна обозревателя.

Замечание

Относительно точности указания высоты кадров в процентах можно не волноваться: если сумма не равна 100%, масштаб кадров будет пропорционально изменен.

Задание параметров кадров в относительных единицах выглядит так:

```
<FRAMESET COLS="*,2*,3*">
```

Здесь символ * означает пропорциональное деление окна программы просмотра. То есть в данном случае окно будет разделено на три вертикальных кадра, первый из которых будет иметь ширину в 1/6, второй — в 2/6 (или 1/3) и третий — в 3/6 (или 1/2) от ширины развернутого окна обозревателя. Единицы при указании относительных значений можно опустить.

Указание значений атрибутов ROWS и COLS может быть и смешанным, например:

```
<FRAMESET COLS="100,25%,*,2*">
```

Здесь первому кадру будет присвоено абсолютное значение в 100 пикселов по ширине, второму — 25% от ширины окна. Оставшееся пространство делится между третьим и четвертым кадрами в пропорции 1:2.

Приоритеты в определении значений атрибутов следующие: в первую очередь слева направо отводится место для кадра с абсолютным значением, затем — для кадра со значением в процентах и в последнюю очередь — для кадров с относительными величинами.

Замечание

При использовании абсолютных значений в атрибутах ROWS и COLS не следует задавать большие кадры — они должны помещаться в окне обозревателя любого размера. Совместно с такими кадрами для лучшей балансировки рекомендуется использовать хотя бы один кадр, определенный в процентах или относительных величинах.

Чтобы разбить окно обозревателя хотя бы на 2 кадра, необходимо создать 3 файла (один файл основного документа, в котором задается кадровая структура, и еще по одному файлу на каждый кадр).

Рассмотрим в качестве примера код из файла main.html, в котором окно разбивается на 3 кадра:

```
<HTML>
<HEAD>
</HEAD>
<FRAMESET ROWS="50%,50%">
<FRAME SRC="file1.htm">
<FRAMESET COLS="25%,75%">
<FRAME SRC="file21.htm"> <FRAME SRC="file22.htm">
</FRAMESET>
</FRAMESET> <NOFRAMES>
Ваш обозреватель не поддерживает кадры.
</NOFRAMES> </HTML>
```

Текст файла file1.htm:

```
<Font Size=10>Первый кадр</Font>
```

Текст файла file21.htm:

```
<Font Size=2>Второй кадр, первый подкадр</Font>
```

Текст файла file22.htm:

```
<Font Size=3>Второй кадр, второй подкадр</Font>
```

Вид окна обозревателя Netscape Navigator, отображающего содержимое файла main.html, приведен на рис. 30.3.

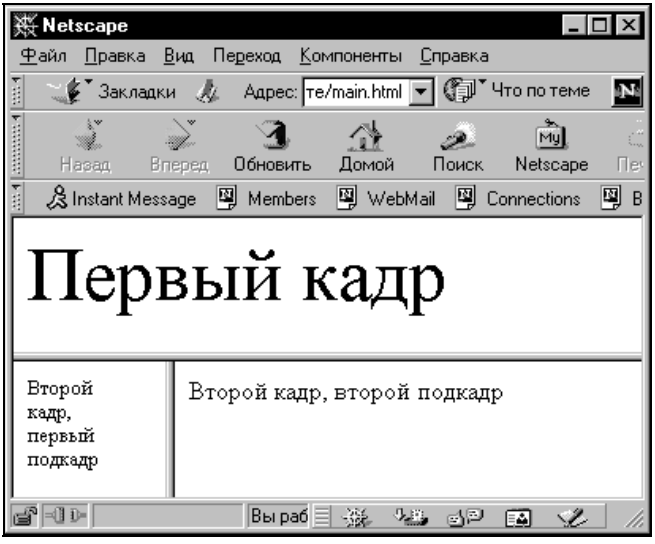


Рис. 30.3. Вид содержимого файла main.html в окне обозревателя

Как видно из рисунка, окно содержит три кадра. Верхний кадр занимает половину окна обозревателя по высоте. Нижняя половина поделена еще на два кадра. На левый нижний кадр (первый подкадр второго кадра) приходится 25% ширины окна обозревателя. Правый нижний кадр (второй подкадр второго кадра) занимает 75% ширины окна обозревателя. Такое разбиение на кадры реализуется путем включения в набор кадров тега `<FRAMESET>` еще одного набора кадров.

Теги создания форм

Формы HTML являются основным средством организации интерактивного взаимодействия в Интернете при разработке Web-приложений. Они служат для пересылки данных от пользователя к Web-серверу. С помощью элементов HTML-документа можно создавать простые формы, предполагающие ответы типа "да" и "нет", а также разрабатывать сложные формы для заказов или для того, чтобы получить от пользователей какие-либо комментарии и пожелания, фиксировать выбор параметра и т. д.

В HTML-документе формы задаются с помощью контейнерного тега <FORM> и состоят из нескольких полей ввода (ограниченная область в окне обозревателя), которые обозреватель отображает в виде графических элементов управления: флажков, переключателей, полей для ввода текста, кнопок и т. д. Для создания полей ввода предназначены теги <INPUT>, <SELECT>, <TEXTAREA>, которые задаются внутри тега <FORM>.

Тег <FORM> в общем виде записывается в следующем формате:

```
<FORM ACTION="URL" METHOD=Метод_передачи ENCTYPE=MIME-тип>
  Содержимое формы
</FORM>
```

Атрибут ACTION является обязательным. В качестве его значения задается адрес URL программы-сценария (модуля расширения сервера на базе интерфейса CGI, ISAPI), которая будет обрабатывать извлеченную из формы информацию.

Атрибут METHOD определяет метод пересылки данных формы от обозревателя к Web-серверу. Если атрибут имеет значение GET, то данные формы передаются в составе запроса URL, будучи присоединенными к нему справа от символа ? в виде пар Переменная=Значение, разделенных символом &. Например, первая строка запроса может иметь следующий вид:

```
GET /bhv.ru/cgi-bin/prog1.cgi?name=stive&format=HTML HTTP/1.0
```

Web-сервер после получения запроса присваивает переменной среды QUERY_STRING значение строки запроса и вызывает программу-сценарий, указанную в начальной части строки запроса. Эта программа может обратиться к переменной среды QUERY_STRING для обработки размещенных в ней данных.

Если атрибут METHOD имеет значение POST, то данные формы пересылаются Web-серверу в теле запроса, который передает их, например, в программу CGI через стандартный ввод.

Значение атрибута ENCTYPE определяет формат кодирования данных (или медиатип) содержимого формы при передаче их от обозревателя к Web-серверу. Кодирование данных выполняется для предотвращения их искажения при передаче.

Теги-дескрипторы <SELECT>, <TEXTAREA> и <INPUT> применяются в HTML для создания интерфейсных объектов внутри формы (тег-дескриптор — это тег, который может использоваться только внутри тега <FORM>).

Замечание

Использование тегов-дескрипторов вне форм поддерживается в обозревателе Internet Explorer. Navigator требует использовать теги-дескрипторы только внутри тега-контейнера <FORM>.

Эти теги имеют следующее назначение:

- <SELECT> — используется для выбора нужной строки в окне с полосой прокрутки либо в раскрывающемся меню;
- <TEXTAREA> — позволяет вводить многострочную текстовую информацию;
- <INPUT> — является многоцелевым тегом и позволяет вводить текстовую информацию в виде строки, создавать интерфейсные объекты в виде флажков (checkbox), переключателей (radiobutton), кнопок и т. п.

Тег <*SELECT*>

Тег <SELECT> является контейнерным и служит для выбора внутри формы одной опции (параметра) из набора без использования полей выбора и переключателей. Тег позволяет также компактно представить элементы выбора в виде раскрывающегося или прокручиваемого списка (меню).

Тег <SELECT> записывается в следующем формате:

```
<SELECT NAME=Имя_поля SIZE=Число MULTIPLE>
  Элементы меню или списка
</SELECT>
```

Атрибут SIZE определяет количество видимых элементов выбора: при SIZE=1 используется раскрывающееся меню (список), при значении SIZE, большем 1, создается прокручиваемый список, для которого Число определяет количество видимых элементов. Задание атрибута MULTIPLE позволяет одновременно выбирать несколько элементов меню или списка.

Элементы меню в теге <SELECT> задаются с помощью тега <OPTION>, записываемого в следующем виде:

```
<OPTION SELECTED VALUE=Значение> Текст
```

Атрибут SELECTED задает выбор элемента по умолчанию. Атрибут VALUE содержит значение, пересылаемое серверу, если данный элемент выбран в меню или списке. Текст, расположенный сразу после тега, выводится в окне обозревателя на месте элемента в списке.

Тег <*TEXTAREA*>

Тег <TEXTAREA> является контейнерным и используется для создания внутри формы поля ввода многострочного текста. Это поле обычно отображается в виде ограниченной прямоугольной области с горизонтальной и вертикальной полосами прокрутки. Тег имеет следующий формат:

```
<TEXTAREA NAME=Имя ROWS=Число COLS=Число>
  Текст по умолчанию
</TEXTAREA>
```

Атрибуты ROWS и COLS определяют число строк и число столбцов видимого текста соответственно. Между открывающим и закрывающим тегами может быть размещен текст, отображаемый в поле ввода по умолчанию.

Тег <*INPUT*>

Тег <INPUT> предназначен для генерирования внутри формы полей для ввода текста, пароля, имени файла и задания различных кнопок. Тег имеет следующий формат:

```
<INPUT TYPE=Тип_поля_ввода NAME=Имя_поля_ввода Дополнительные_атрибуты>
```

Первые два атрибута являются обязательными. Атрибут TYPE определяет тип поля ввода: кнопка выбора, кнопка отправки и др. Атрибут NAME задает имя поля, которое используется как идентификатор передаваемого Web-серверу значения. Состав дополнительных атрибутов зависит от значения атрибута TYPE, определяющего тип поля.

Атрибут TYPE тега <INPUT> может принимать следующие значения:

- TEXT — является значением по умолчанию. В этом случае создается интерфейсный элемент в виде одной строки для ввода данных. Для такого типа поля ввода наиболее часто используется дополнительный атрибут VALUE, который предоставляет доступ к данным, находящимся внутри поля;
- PASSWORD — в этом случае создается интерфейсный элемент, аналогичный предыдущему, но вводимые в него символы пароля заменяются звездочками;
- CHECKBOX — создается флажок, который позволяет выбрать сразу несколько из предложенных опций. Для этого значения может использоваться атрибут CHECKED, который определяет установленный по умолчанию флажок;
- RADIO — создается переключатель, позволяющий выбрать только одно значение;
- RESET — создается кнопка для обновления формы;
- SUBMIT — создается кнопка, по нажатии которой введенные данные отправляются на сервер для обработки программой-сценарием.

В качестве примера использования тега <INPUT> и других тегов формы приведем следующий код (листинг 30.1).

Листинг 30.1. Пример использования тегов формы

```
<FORM>
Введите ваше имя:
<INPUT TYPE="text" NAME="My" Value="Имя" SIZE="10" MAXLENGTH="20">
Введите пароль:
<INPUT TYPE="password" NAME="pass1" Size="30" MAXLENGTH="30">
<BR>
Выберите тип действия: <BR>
<INPUT TYPE="checkbox" NAME="mycheckbox1" VALUE="Параметр1" CHECKED>
Действие 1
<INPUT TYPE="checkbox" NAME="mycheckbox2" VALUE="Параметр2">
Действие 2
<BR>
<INPUT TYPE="radio" NAME="my1" VALUE="Параметр3"> Действие 3
<INPUT TYPE="radio" NAME="my2" VALUE="Параметр4"> Действие 4
<BR>
<TEXTAREA NAME="My_textarea " ROWS=4 COLS=40>
Справочный текст, внутри которого <BR>
можно ввести собственные данные <BR>
Эта форма посылает данные серверу
</TEXTAREA> <BR>
<SELECT NAME="menu">
<OPTION SELECTED VALUE="Действие1"> Действие1
<OPTION VALUE="Действие2"> Действие2
<OPTION VALUE="Действие3"> Действие3
</SELECT>
<INPUT TYPE="reset" VALUE="Reset">
<BR>
<INPUT TYPE="submit" VALUE="Послать данные!">
</FORM>
```

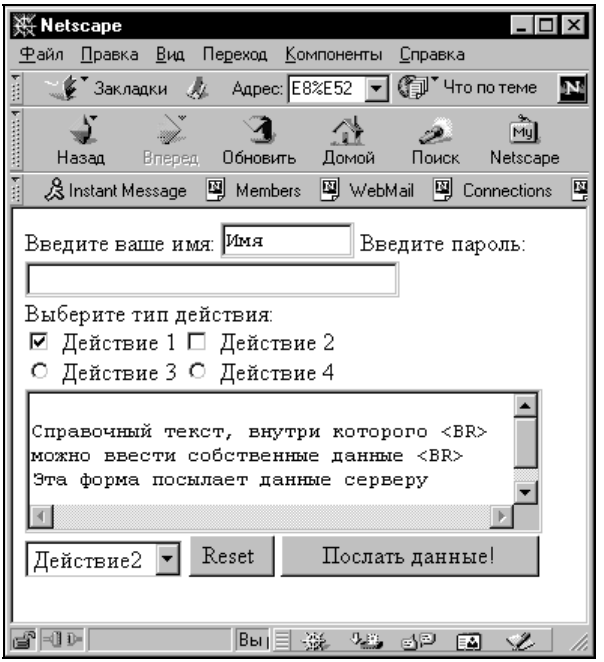


Рис. 30.4. Простая форма, отображенная в окне обозревателя

Результат отображения приведенного кода в окне обозревателя показан на рис. 30.4.

В этом примере элемент `text` снабжен дополнительными атрибутами: `SIZE="10"` определяет размер рамки для размещения 10 символов, а `MAXLENGTH="20"` — максимальный размер внутреннего представления текстовой строки в символах. Атрибут `NAME` у всех типов используется для получения доступа к заданному элементу HTML-документа.

Графические теги

В HTML-документе используются два формата графических изображений (GIF и JPEG), которые понимают большинство современных обозревателей.

Для воспроизведения изображения на HTML-странице применяется тег `<IMG>`:

```
<IMG Атрибут1=Значение1 ... АтрибутN=ЗначениеN>
```

Рисунок отображается в том месте документа, где находится тег `<IMG>`. Перечислим некоторые наиболее важные атрибуты этого тега.

Атрибут `SRC` используется для указания URL-адреса файла, содержащего рисунок. Например:

```
<IMG SRC="my.gif">
```

С помощью атрибутов `HEIGHT` и `WIDTH` устанавливаются высота и ширина изображения в пикселах. Например:

```
<IMG SRC="my.gif" HEIGHT=200 WIDTH=100>
```

Замечание

Использовать эти атрибуты нужно весьма аккуратно, т. к. при изменении масштаба изображения рисунок может сильно исказиться в обозревателе.

Атрибут ALT позволяет выводить краткое текстовое описание вместо рисунка в случае, если пользователь отменил загрузку изображения. Если же изображение выводится, то это описание отображается на месте рисунка до начала его загрузки. Например:

```
<IMG SRC="my.gif" ALT="Описание рисунка">
```

Атрибут BORDER задает ширину рамки (в пикселах) вокруг рисунка.

Теги задания ссылок

Одним из важных элементов HTML-документа является ссылка. Она состоит из *указателя* (anchor) и *адресной части* (URL-адрес). Указатель ссылки виден в окне обозревателя при отображении HTML-документа. При выборе пользователем указателя ссылки обозреватель загружает HTML-документ, находящийся по соответствующему URL-адресу. Указателем ссылки может быть текст или изображение. Обычно обозреватель отображает указатель ссылки в виде подчеркнутого прямой линией текста.

Ссылки задаются с помощью тега <A>...</A> контейнерного типа:

```
<A HREF="URL-адрес"> Текст указателя или Теги, задающие изображение </A>
```

Атрибут HREF тега <A> определяет URL-адрес HTML-документа, который отображается в окне обозревателя при выборе ссылки.

Информация, находящаяся внутри контейнера, может содержать текст указателя или специальные теги, задающие изображение. Пример текстового указателя:

```
<A HREF="my.html"> Для отображения файла my.html выберите этот текст </A>
```

Для создания графической ссылки следует в контейнер тега ссылки включить тег, определяющий изображение, например:

```
<A HREF="my.html"> <IMG SRC="рисунок.gif"> </A>
```

Если необходимо ссылаться на разные части текущего документа, то применяются *внутренние* ссылки. Для задания внутренней ссылки нужно в атрибуте HREF тега <A> поместить имя указателя со специальным префиксом #:

```
<A HREF="#Имя"> Текст ссылки </A>
```

где Имя — специальное имя, которое используется для организации связи с фрагментом текста; Текст ссылки — текст указателя ссылки, отображаемый в окне обозревателя. Например:

```
<A HREF="#Next"> Переход к следующему фрагменту текста </A>
```

Для указания места перехода по внутренней ссылке используют следующий тег:

```
<A NAME=Имя> Фрагмент текста, к которому осуществляется переход </A>
```

Здесь Имя — имя указателя, которое входит в атрибут HREF тега внутренней ссылки. Например:

```
<A HREF="#Go"> Текст для перехода по внутренней ссылке </A>
...
<A NAME=Go> Фрагмент текста, к которому осуществляется переход
            по внутренней ссылке </A>
```

В последнее время широкое распространение получил расширяемый язык разметки XML. У этого языка есть средства структурированного представления данных и доступа к ним, базирующиеся на принципах, применяемых в СУБД. Поэтому его можно использовать для публикации сильно структурированных данных (в частности, баз данных) на HTML-страницах. В следующем разделе мы рассмотрим некоторые особенности языка XML.

Использование XML

Расширяемый язык разметки XML представляет собой дальнейшее развитие языка HTML и по сравнению с ним обеспечивает ряд дополнительных возможностей:

- подготовка и настройка XML-документов со сложной структурой;
- использование информационных объектов (entity) для работы с группами данных;
- описание типа документа DTD (Document Type Definition);
- контроль правильности документа;
- использование XML-формата для хранения структурированных однотипных данных (ранее для этой цели применялись только БД).

Отметим, что XML принадлежит подмножеству стандартного обобщенного языка разметки SGML (Standard Generalized Markup Language). Хотя язык SGML предоставляет более развитые возможности по сравнению с XML, его применение требует много времени на подготовку и обработку соответствующего SGML-документа. Поэтому он не получил такого широкого распространения, как XML или HTML.

Главное отличие XML от HTML заключается в том, что с помощью XML можно не только наполнить создаваемый документ размеченным содержимым, но и определить структуру документа и типы хранимых в нем данных. Средства XML обеспечивают стандартизованное представление данных, облегчая этим задачу использования (импорта) данных из других источников.

Документы XML можно разделить на правильно построенные (well-formed) и действительные (valid). *Правильно построенные* документы XML удовлетворяют спецификации XML, но не имеют определения типа документа DTD. *Действительные* документы XML, в отличие от правильно построенных, содержат также определение типа документа DTD (определяющее типы используемых данных и возможную структуру документа) или ссылаются на него.

Спецификацию XML поддерживает ряд современных программ просмотра, в частности, обозреватели Internet Explorer версии 4.0 и выше и Netscape Navigator версии 5.0 и выше.

При описании XML-документа накладываются следующие дополнительные (по сравнению с HTML) ограничения:

- каждому открывающему тегу должен соответствовать закрывающий тег;
- пары открывающих и закрывающих тегов могут вкладываться друг в друга, однако пересечение пар открывающих и закрывающих тегов не допускается;
- при определении элементов и атрибутов учитывается регистр символов;
- все значения атрибутов должны указываться в апострофах (`''`) или кавычках (`""`).

Рассмотрим более подробно состав XML-документа, а также назначение, объявление и примеры использования его составляющих.

Состав XML-документа

В XML-документе обычно присутствуют: элементы, атрибуты, информационные объекты и комментарии. *Элементы* представляют собой части документа. Элемент объявляется так:

```
<!ELEMENT Имя Содержание>
```

Объявление элементов может размещаться внутри документа или в определении типа документа DTD. В DTD можно объявить произвольное число элементов, при этом не все из них должны использоваться в документе XML. Отдельные элементы могут содержать другие элементы. Например, чтобы указать, что элемент `employee` (сотрудник) содержит элементы `surname` (фамилия) и `post` (должность), нужно поместить в DTD определения элементов в следующем порядке:

```
<!ELEMENT surname(#PCDATA)>
<!ELEMENT post(#PCDATA)>
<!ELEMENT employee(surname, post)>
```

Здесь строка `#PCDATA` является *атрибутом* и указывает, что соответствующий элемент содержит символьные данные.

Другие символы служат для задания режима использования элемента, например, определяют, что элемент в содержании документа необязателен или может использоваться несколько раз. Основные символы, применяемые при описании элемента, приведены в табл. 30.1.

Таблица 30.1. *Основные символы в описании элемента*

Символ	Имя символа	Назначение
`()`	Круглые скобки	Содержит альтернативные элементы или элементы списка
`|`	Вертикальная линия	Разделение альтернативных элементов
`?`	Знак вопроса	Элемент может отсутствовать или использоваться один раз
`,`	Запятая	Разделение элементов в списке
`*`	Звездочка	Элемент может отсутствовать или использоваться любое число раз
`+`	Плюс	Элемент должен использоваться не менее одного раза

Например, в строке

```
<!ELEMENT employee(surname, post+)>
```

знак + указывает на то, что в составе элемента `employee` элемент `post` должен присутствовать не менее одного раза.

Атрибуты служат для уточнения характеристик элементов, вида содержащейся в них информации и порядка ее размещения. Атрибуты можно указывать в строке объявления элемента (как показано выше) или в описании DTD (в задаваемом с помощью тега `<!ATTLIST>` списке атрибутов). Для действительного документа XML объявление атрибута в DTD задается так:

```
<!ATTLIST Имя_элемента Имя_атрибута Тип Значение_по_умолчанию>
```

Атрибут элемента может быть строковым, маркированным или перечислимым и иметь следующие типы и назначение:

- `CDATA` (символьные данные);
- `Enumerated` (набор возможных значений);
- `ID` (уникальный идентификационный номер);
- `IDREF` (указатель на элемент с заданным значением `ID`);
- `IDREFS` (указатель на несколько элементов в документе в виде номеров `ID`, разделенных пробелами);
- `ENTITY` (имя внешнего информационного объекта);
- `ENTITIES` (указатель на несколько внешних информационных объектов в виде имен `ENTITY`, разделенных пробелами);
- `NMTOKEN` (имя XML);
- `NOTATION` (имя в обозначении, объявленном в DTD);
- `NMTOKENS` (множество имен XML, разделенных пробелами).

Предположим, что в документе содержится тег вида:

```
<greeting language="Russia">
```

Приведенный тег может быть объявлен в DTD так:

```
<!ELEMENT greeting #PCDATA
<!ATTLIST greeting language CDATA "English"
```

Здесь в первой строке указывается, что элемент `greeting` может содержать управляемые символьные данные. Во второй строке для этого элемента задается атрибут `language` типа `CDATA`, и для атрибута устанавливается значение `English`, используемое по умолчанию.

Комментарии в XML, как и в HTML, пишутся в обрамлении открывающего `<!--` и закрывающего `-->` тегов.

Информационные объекты

Обычно документ XML состоит из двух частей: пролога и данных. *Пролог* включает в себя объявления XML и описание типа документа DTD, а *данные* представляют собой

размеченный текст. В общем случае документ XML может содержать данные и объявления из различных источников: баз данных, сценариев и др., т. е. документ XML состоит из пролога и следующих за ним *информационных объектов* (entity). В качестве информационного объекта может выступать файл, запись базы данных или другой элемент данных.

Информационные объекты позволяют задавать имена для наборов символьных данных или файлов так, чтобы их можно было включать в состав XML-документа. Например, если в документе многократно используется символьная строка с информацией об издательстве, то целесообразно создать информационный объект с помощью следующего кода:

```
<!ENTITY redBHV "СПб.: БХВ-Петербург"
```

Теперь в документе достаточно указывать ссылки вида `&redBHV`, синтаксический анализатор при обработке документа заменит их на строку `СПб.: БХВ-Петербург`.

Приведенный пример показывает использование *внутреннего* информационного объекта, описание которого содержится в самом документе. Кроме того, допустимо использовать *внешний* информационный объект, который включает имя и ссылку на контейнер, хранящий его содержание. Объявление внешних информационных объектов выполняется с помощью инструкции следующего формата:

```
<!ENTITY name (PUBLIC | SYSTEM) "content")
```

Здесь строка `name` задает имя информационного объекта, а ключевые слова `PUBLIC` или `SYSTEM` указывают на доступность использования информационного объекта в документах.

В XML имеется 5 *предопределенных* общих ссылок: `<`, `>`, `&apos;`, `&quote;` и `&`, которые ссылаются на текстовые информационные объекты (символы) <, >, ', " и & соответственно. Они применяются в документах XML, чтобы избежать неправильной интерпретации соответствующих символов как части разметки документа. Например, символы < и > при их непосредственном использовании могут интерпретироваться как начало и конец тега.

Определение типа документа

Создание отдельного определения типа документа DTD для каждого документа XML обеспечивает:

- возможность проверять корректность документа для синтаксического анализатора;
- удобство задания и изучения структуры документа определенного типа;
- возможность применять не только элементы и атрибуты, но и информационные объекты.

В состав DTD входят пять частей:

- типы документов;
- элементы;
- атрибуты;
- информационные объекты;
- комментарии.

Определение типа документа может размещаться внутри документа XML, в этом случае говорят о *внутреннем* DTD, или быть задано во внешнем файле (*внешнее* DTD). Применительно к внешним DTD различают системное (`SYSTEM`) и общее (`PUBLIC`) определения типа документа. Системное DTD может использоваться только одним или несколькими документами XML, а общее DTD — любым документом XML.

Одно из важнейших назначений DTD заключается в определении структуры документа как иерархии составляющих его элементов. В частности, применительно к книге соответствующее внутреннее DTD и сам документ XML могут быть заданы так (листинг 30.2).

Листинг 30.2. Пример определения типа документа

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book [
<!ENTITY % phras "PCDATA | emphasis">
<!ELEMENT book (title_page, content)>
<!ELEMENT title_page (author, title)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT content (part, chapter+)>
<!ELEMENT part (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
]>
<book>
<title_page>
<author> Ivanov Dm. </author>
<title> Name of book </title>
</title_page>
<content>
<part> Name of Part </part>
<chapter> Text of Chapter1 </chapter>
<chapter> Text of Chapter2 </chapter>
<chapter> Text of Chapter3 </chapter>
</content>
</book>
```

Здесь первая строка кода указывает, что это XML-документ, и определяет используемую систему кодирования — 8-битовую кодировку Unicode, соответствующую символам ASCII. Тег `<!DOCTYPE book [` указывает на начало описания DTD с именем `book`, а тег `]>` — на окончание DTD. В строке

```
<!ELEMENT content (part, chapter+)>
```

знак `+` говорит о том, что элементов `chapter` (глава) в родительском элементе `content` (содержание) книги может быть несколько.

При обнаружении ошибок в исходном коде документа XML обозреватель выдает сообщение о характере и местоположении первой из них, как показано, например, на рис. 30.5. Отметим, что при подготовке документов HTML возможность проверки синтаксических ошибок отсутствует в принципе.

Для приведенного документа XML и его описания DTD в окне обозревателя Microsoft Internet Explorer 5.0 будет отображаться структура, изображенная на рис. 30.6.

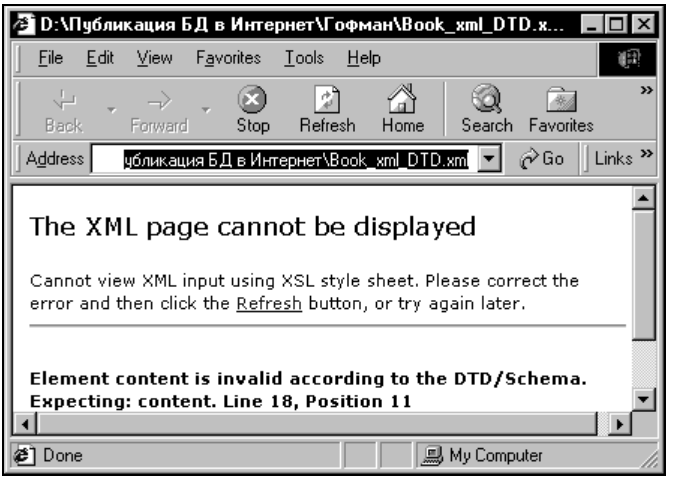


Рис. 30.5. Сообщение об ошибке в окне обозревателя

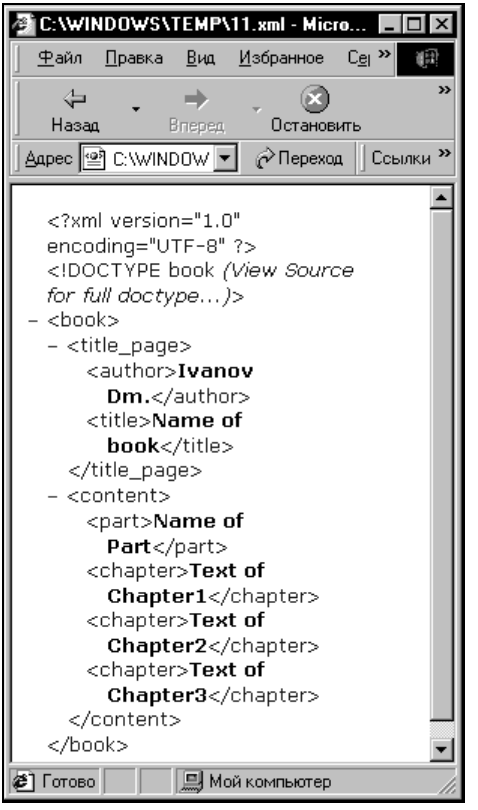


Рис. 30.6. Вид структуры в окне обозревателя

Для создания *внешнего* DTD следует поместить объявления всех размещаемых в нем элементов, атрибутов и информационных объектов в отдельный файл с расширением dtd, например, в файл book.dtd. В состав внешнего DTD нужно поместить также инструкцию по обработке кода XML, включив строку

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Чтобы получить доступ к внешнему DTD, размещенному в сети, в XML-документ нужно включить следующее описание:

```
<!DOCTYPE book SYSTEM "http://www.xom.com/dtds/book.dtd"
```

Здесь `http://www.xom.com/dtds/book.dtd` — адрес размещения файла DTD в сети. Если DTD является общим, в приведенной строке вместо слова `SYSTEM` нужно указать `PUBLIC`.

XML как средство обмена данными

Как отмечалось, расширяемый язык разметки XML представляет собой развитие языка HTML и по сравнению с ним обеспечивает ряд дополнительных возможностей. Главное отличие XML от HTML заключается в том, что он позволяет определить структуру документа и типы хранимых в нем данных.

Напомним, что одно из достоинств XML состоит в том, что в разрабатываемых с его помощью документах описание структуры хранимых данных отделено от собственно данных. В связи с этим XML представляет собой удобное средство обмена данными между отдельными приложениями, т. к. позволяет обеспечить согласованный обмен данными в случаях, когда структуры данных (например, имена и типы полей) в приложениях различаются.

Кроме того, с помощью XML можно упростить доступ к данным, хранимым в базах данных. Например, для доступа к данным персональных БД или табличного процессора Excel пользователю требуется установить соответствующие программные инструменты. Вместо этого можно создать активные серверные страницы (ASP) или сценарии на языке JScript или VBScript, которые будут извлекать данные из БД и помещать их в документ XML. В дальнейшем информацию из полученного таким образом документа XML можно использовать в других приложениях или отображать на Web-страницах, т. е. полученные данные становятся доступными для всех пользователей, имеющих обозреватель, независимо от наличия СУБД или табличного процессора.

Документы XML могут использоваться как на стороне клиента, так и на стороне сервера. Применение XML-документов на стороне *клиента* в настоящее время сдерживается в основном отсутствием соответствующих инструментов: не все обозреватели поддерживают просмотр и работу с документами XML, хотя в большинстве современных продуктов такая возможность уже имеется. Как уже говорилось, к ним относятся обозреватели Microsoft Internet Explorer, начиная с версии 4.0, и Netscape Navigator версии 5.0 и выше.

При работе с документами XML на *стороне сервера* используются языковые средства, обычно применяемые для расширения возможностей сервера, такие как Java и JScript. Отметим еще, что последние версии современных систем программирования, ориентированных на разработку Web-приложений, также поддерживают средства создания и

обработки документов XML. В частности, такие средства имеются в составе JBuilder 4.0 и Delphi 7.

Более того, в системе Delphi 7 XML является языком кодирования данных, передаваемых при использовании Web-служб с помощью протокола SOAP, объединяющего XML и протокол HTTP и обеспечивающего тем самым универсальность обмена данными.

Программа XML Mapper

Программа XML Mapper (XML-преобразователь) задает отображение (определяет соответствие) между XML-документами и пакетами данных, используемыми клиентскими наборами данных. Эту программу можно запустить из среды Delphi командой **Tools | XML Mapper** или запуском файла xmlmapper.exe, находящегося в каталоге BIN главного каталога Delphi, а также через меню **Пуск** Windows, выбрав команду **Программы | Borland Delphi 7 | XML Mapper**.

Пакет данных содержит метаданные, описывающие структуру XML-документа. *Отображение* (трансформация) представляет собой соответствие между элементами XML-документа и полями в пакете данных. С помощью программы XML Mapper можно задать отображения:

- существующей XML-схемы или документа на клиентский набор данных;
- существующего пакета данных на новую XML-схему;
- существующей XML-схемы на существующий пакет данных.

Заданное отображение можно сохранить в трансформационном файле (файле соответствия) с расширением xtr. Обычно трансформационные файлы создаются в паре: для прямого и обратного преобразования. Полученные с помощью программы XML Mapper трансформационные файлы могут использоваться компонентами типа `TXMLTransform`, `TXMLTransformProvider` и `TXMLTransformClient` для выполнения преобразования данных в приложении.

Окно программы XML Mapper состоит из трех панелей (рис. 30.7):

- панель документа (**Document**) — отображает XML-документ и 3 варианта его схем;
- панель отображения (**Transformation**) — задает соответствие между XML-документом и пакетом данных;
- панель пакета данных (**Datapacket**) — отображает метаданные пакета данных.

Панель документа содержит две вкладки, позволяющие просматривать содержимое открытого (загруженного) XML-документа в виде дерева элементов (вкладка **Document View**) и в виде схем DTD, XDR или XSD (вкладка **Schema View**). Открытие документа выполняется командой **File | Open** главного меню программы или командой **Open XML Document** контекстного меню панели документа.

Замечание

Файлы с расширениями dtd (Data Type Definition), xdr (reduced XMLData) и xsd (XML schema) служат для хранения соответствующих вариантов схем XML-документа, отображаемых на вкладке **Schema View**.

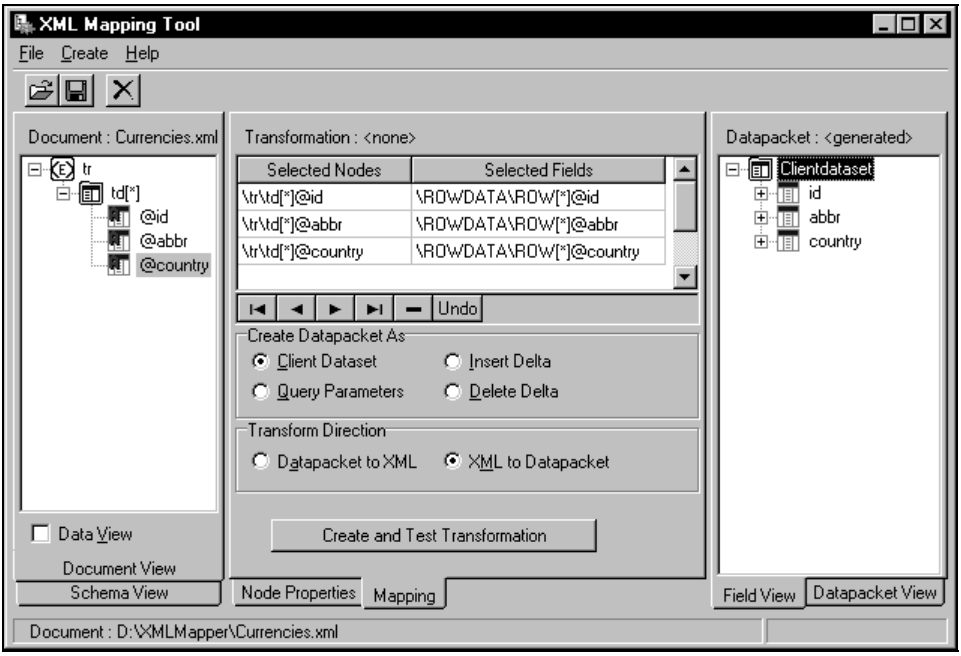


Рис. 30.7. Главное окно программы XML Mapper

Панель пакета данных содержит две вкладки, позволяющие просматривать содержимое текущего пакета данных в виде дерева полей (вкладка **Field View**) и структуры (вкладка **Datapacket View**). Открытие пакета данных выполняется командой **File | Open** главного меню программы или командой **Open Datapacket** контекстного меню панели пакета данных. Файл пакета данных имеет расширение xml, кроме того, можно открывать пакет данных, сохраненный в двоичном формате (файл с расширением cds).

Панель отображения определяет соответствие между элементами XML-документа и полями пакета данных. Эта панель содержит две вкладки: **Node Properties** и **Mapping**.

Вкладка **Node Properties** позволяет просматривать и изменять свойства элемента (Node), выбранного в открытом XML-документе. Свойства элементов открытого документа можно сохранить в файле с расширением xrp. Сохранение свойств выполняется командой **File | Save | Repository** главного меню программы или командой **Save Repository** контекстного меню панели отображения. Впоследствии сохраненные свойства элементов XML-документа можно загрузить из файла .xrp.

Свойства выбранного элемента отображаются в таблице, содержащей названия свойств (столбец **Properties**) и значения свойств (столбец **Values**). Отметим следующие свойства:

- `Attribute Name` (имя элемента);
- `Mapped Name` (имя поля в пакете данных, которому соответствует элемент);
- `Data Type` (тип данных, например, `String`, `Float` или `Integer`);
- `Data Format` (формат данных);
- `Max Length` (максимальная длина (для строковых данных));

- `Value Required` (требование обязательности значения);
- `In Primary Key` (признак включения поля в состав первичного ключа пакета данных);
- `Default Value` (значение по умолчанию);
- `Sample Value` (пример значения).

Первоначально свойства имеют значения по умолчанию, определяемые на основании информации из XML-документа. Эти значения можно изменять, например, задать другой тип данных.

На вкладке **Mapping** находится таблица, отображающая соответствие между элементами XML-документа (столбец **Selected Nodes**) и полями пакета данных (столбец **Selected Fields**). Добавление нового элемента к таблице выполняется двойным щелчком на нем или командой **Select** контекстного меню элемента. Команда **Select All** включает в таблицу все элементы документа. Под таблицей соответствия расположен Навигатор, позволяющий перемещаться по элементам, а также удалять элементы из таблицы.

Создание пакета данных из XML-документа и, наоборот, получение XML-документа из пакета данных выполняется, соответственно, командами **Create | Datapacket from XML** и **Create | XML from Datapacket** главного меню или командами **Create Datapacket from XML** и **Create XML from Datapacket** контекстных меню панелей документа и пакета данных. В листинге 30.3 приводится текст файла Currencies.xml XML-документа и созданного на его основе файла пакета данных CurrenciesDatapacket.xml.

Листинг 30.3. Пример XML-документа и файла пакета данных

Файл Currencies.xml:

```
<?xml version="1.0" standalone="yes"?>
 <tr>
   <td id="1" abbr="CHF" country="Swiss"/>
   <td id="2" abbr="RUB" country="Russia"/>
   <td id="3" abbr="GBP" country="Great Britain"/>
   <td id="4" abbr="USD" country="USA"/>
 </tr>
```

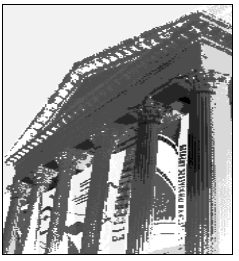
Файл CurrenciesDatapacket.xml:

```
<?xml version="1.0" standalone="yes"?>
<DATAPACKET Version="2.0">
<METADATA>
  <FIELDS>
    <FIELD attrname="id" fieldtype="string" WIDTH="1"/>
    <FIELD attrname="abbr" fieldtype="string" WIDTH="3"/>
    <FIELD attrname="country" fieldtype="string" WIDTH="13"/>
  </FIELDS>
  <PARAMS/>
```

```
</METADATA>
<ROWDATA/>
</DATAPACKET>
```

Отображение (трансформация) XML-документа на пакет данных и обратное преобразование выполняются командой **Create | Transformation** главного меню или командой **Create Transformation** контекстного меню панели отображения. Преобразование также можно выполнить нажатием кнопки **Create and Test Transformation**, при этом направление преобразования задается переключателями группы **Transform Direction**. Созданное отображение можно сохранить в трансформационном файле командой **File | Save | Transformation** главного меню или командой **Save Transformation** контекстного меню панели отображения.

ГЛАВА 31



Характеристика Web-приложений

Современные Web-приложения применяются для создания информационных систем в сетях Интернет/интранет и обычно строятся как многозвенные (многоуровневые) приложения, публикующие БД.

При большом числе клиентских компьютеров многоуровневая архитектура позволяет уменьшить нагрузку на сервер баз данных и линии связи. В этой главе описаны принципы функционирования и структура Web-приложений, а также особенности Web-приложений, публикующих БД в сетях Интернет/интранет.

Принципы функционирования Web-приложений

Развитие сетей Интернет/интранет привело к появлению новой категории программ — Web-приложений. К Web-приложениям относят набор Web-страниц, сценариев и других программных средств, расположенных на одном или нескольких компьютерах (клиентских и серверных) и объединенных для выполнения прикладной задачи. При этом Web-приложения, публикующие БД в Интернете, представляют собой отдельный класс Web-приложений.

Современные информационные системы, построенные на основе Web-приложений, использующих БД, базируются на многоуровневой клиент-серверной архитектуре и принципах функционирования Интернета. С основами этой архитектуры мы ознакомились в предыдущей главе. Здесь мы кратко повторим, как осуществляется работа приложений в Интернете.

Web-приложения выполняются на стороне Web-сервера, который находится на Web-узлах сети Интернет. Web-сервер обрабатывает запросы обозревателя на получение Web-страниц и отсылает ему требуемые данные в формате Web-документов.

Обмен данными в Интернете осуществляется на основе коммуникационного протокола TCP/IP и протокола более высокого уровня (приложений) HTTP. Упрощенная схема функционирования Web-приложения приведена на рис. 31.1.

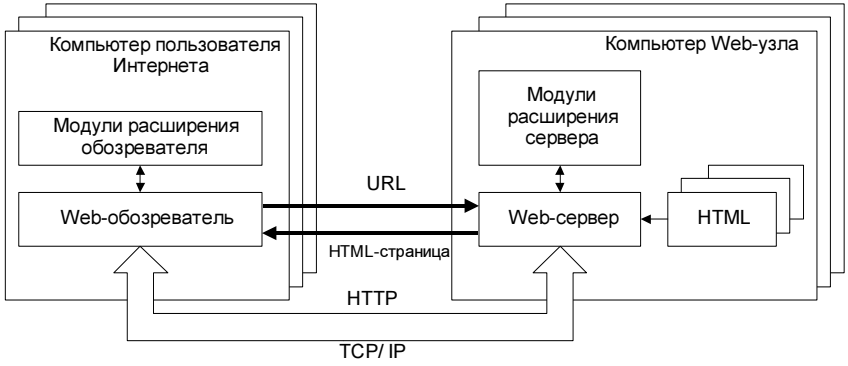


Рис. 31.1. Схема функционирования Web-приложения с модулями расширения

Напомним, что под Web-документом (Web-страницей) понимаются используемые в Интернете документы в форматах HTML, XML, шаблоны в форматах ASP, HTX и т. д.

Для доступа к Web-страницам используются специальные клиентские программы — Web-обозреватели, находящиеся на компьютерах пользователей Интернета. Обозреватель формирует запрос на получение требуемой страницы или другого ресурса с помощью адреса URL. Функции обозревателя заключаются в отображении Web-страниц, сгенерированных сервером или модулями расширения, и отправке запросов пользователя Web-приложению, т. е. обозреватель является связующим звеном между пользователем и Web-приложением. При этом Web-обозреватель устанавливает соединение с требуемым Web-узлом, используя различные протоколы передачи данных, в частности уже рассмотренный протокол HTTP.

Web-приложения в сетях интранет

Развитие технологий глобальной сети Интернет привело к появлению архитектурных и технологических решений для локальных корпоративных сетей, называемых также сетями *интранет* (или интрасетями). В общем случае под сетью интранет понимают выделенную часть Интернета, в которой выполняется Web-приложение (информационная система).

Применение интернет-технологий в корпоративных интрасетях позволяет повысить эффективность функционирования сетей и используемых в них информационных систем. Приложения, построенные на основе интернет-технологий, характеризуются надежностью и масштабируемостью, открытостью архитектуры, простотой освоения и использования.

Кроме того, использование сетей интранет характеризуется значительным снижением затрат на обслуживание, модернизацию и наращивание сети по сравнению с традиционными корпоративными сетями, построенными на клиент-серверных технологиях. Важным достоинством интрасетей является возможность развертывания корпоративных локальных и глобальных сетей на существующей инфраструктуре.

При использовании Web-приложений в сети интранет может использоваться архитектура, показанная на рис. 31.2. Интрасети в общем случае имеют различные внутренние структуры, причем они могут и не иметь выхода в Интернет.

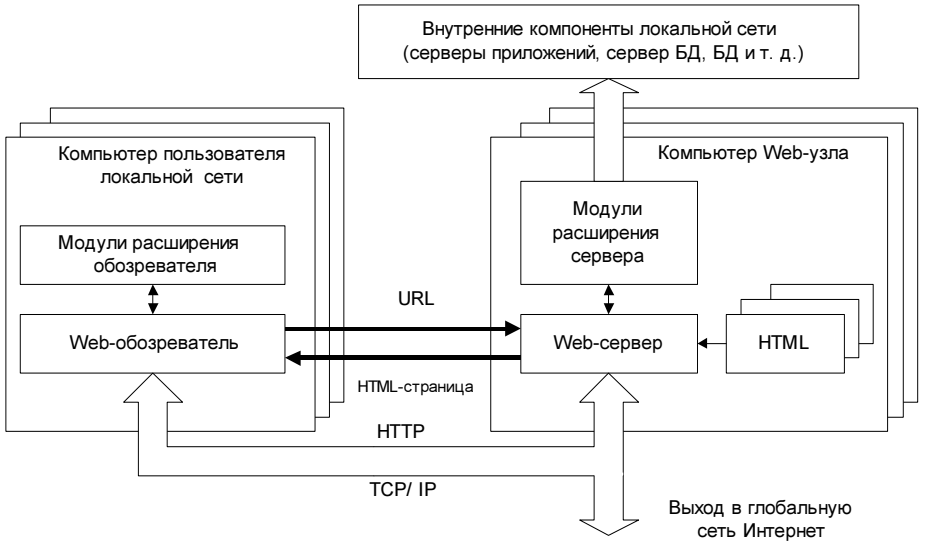


Рис. 31.2. Схема функционирования Web-приложения с модулями расширения

В качестве клиентских приложений в этой архитектуре выступают Web-обозреватели, которые обращаются с запросами к серверу БД или к серверу приложений через Web-сервер. В зависимости от используемой конфигурации Web-сервер может находиться на сервере БД или на сервере приложений.

В функции Web-сервера в сети интранет входит обработка запросов Web-обозревателей на получение информации из разделяемых БД, преобразование этих запросов (может выполняться модулями расширения Web-сервера) в SQL-запросы или другие форматы, понятные для сервера БД или сервера приложений.

Кроме того, интранет-приложение предоставляет следующие дополнительные возможности.

- Удаленные доступ и управление. Концепция *удаленного доступа* подразумевает возможность подключения к интранет-сети извне, т. е. с любого компьютера из Интернета. Под *удаленным управлением* понимается подключение к локальной сети и выполнение функциональных операций по управлению ее ресурсами с удаленного компьютера. Для реализации удаленного управления необходимо наличие специального сервера удаленного доступа и специального программного обеспечения на удаленном компьютере.
- Выход в Интернет клиентов сети. При этом становятся доступными услуги, предоставляемые глобальной сетью: получение актуальной информации в различных сферах, электронная почта, обмен данными с внешними источниками, использование приложений, находящихся в Интернете, и т. д.

Сеть интранет может быть построена на основе использования Web-сервера в локальной сети или на основе услуг, предоставляемых внешним (интернетовским) Web-сервером. Такие услуги, как правило, предоставляет интернет-провайдер, обеспечивающий возможность использования функций своего Web-сервера. Многие компании

используют совмещенную структуру Web-серверов, при которой сама локальная сеть строится с использованием собственного Web-сервера, а выход в глобальную сеть осуществляется через Web-сервер провайдера.

Применение архитектуры Интернета в сетях интранет по сравнению с традиционными архитектурами локальных сетей дает следующие преимущества: стандартизация пользовательского интерфейса; более удобное администрирование и конфигурирование; удешевление установки и лицензирования клиентских компьютеров пользователей.

Для расширения возможностей клиентской части (обозревателя) и серверной части разрабатывают модули расширения обозревателя и сервера, используемые для динамического управления интерфейсными объектами (компонентами) Web-документа.

В следующих разделах мы рассмотрим взаимодействие Web-приложения с дополнительными компонентами: модулем расширения клиентской части и модулем расширения серверной части (см. рис. 31.1).

Web-приложения с модулями расширения серверной части

Архитектура Web-приложений с модулями расширения сервера (рис. 31.3) может включать в себя стандартные модули расширения — DLL-библиотеки, реализующие, например, технологии ASP, IDC/HTX, объекты ActiveX и пр. Кроме того, допускается подключение дополнительных модулей, разработанных с использованием интерфейсов CGI, ISAPI и др.

В этом случае в функции Web-сервера входят обработка запросов Web-обозревателей пользователей сети, вызов (загрузка) соответствующего модуля расширения сервера и передача ему параметров запроса. В результате обработки запроса модулями расширения сервера формируется Web-документ с использованием различных HTML-шаблонов. Готовый Web-документ Web-сервера отсылается обратно Web-обозревателю в формате протокола HTTP.

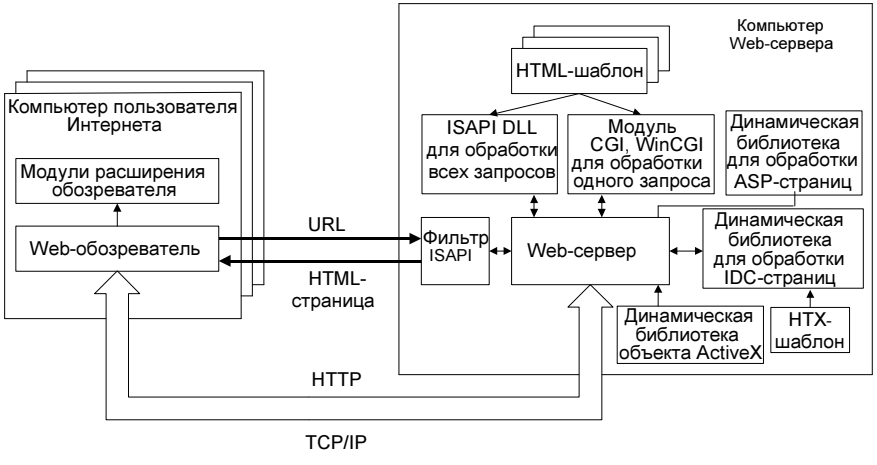


Рис. 31.3. Архитектура Web-приложения с модулями расширения сервера

Как отмечалось, различают пассивное и активное состояния Web-сервера. Так, Web-сервер находится в пассивном состоянии, если формируемый им документ содержит статическую информацию, т. е. на Web-странице отсутствуют средства ввода и обработки запросов к серверу.

Если Web-документ динамически создается в ответ на запрос пользователя (рис. 31.3) или если в обозреватель загружены различные интерактивные визуальные элементы управления, то сервер является активным. Для публикации БД основной интерес представляет как раз активный Web-сервер, который реализуется с помощью модулей расширения.

Для создания модулей расширения Web-сервера используются интерфейсы CGI, WinCGI или API.

Интерфейс *CGI* является стандартным протоколом взаимодействия между Web-сервером и модулями расширения, которые могут применяться для выполнения дополнительных функций, не поддерживаемых сервером. Напомним, что интерфейсу CGI соответствуют обычные консольные приложения операционной системы DOS. Для написания CGI-программ подходит практически любой язык программирования, обеспечивающий доступ к переменным среды и ввод/вывод через стандартные потоки STDIN и STDOUT. В частности, для написания CGI-программ подходят среды разработки Delphi и C++ Builder.

Для запуска CGI-модуля обозреватель должен сформировать запрос к серверу с указанием адреса URL этого модуля. Это можно сделать следующими способами:

- задать URL модуля CGI в адресной строке обозревателя;
- послать серверу запрос на выполнение CGI-модуля путем выбора ссылки, атрибут `HREF` которой содержит адрес этого модуля;
- нажать в форме кнопку типа `Submit`, у которой атрибут `Action` содержит URL-адрес CGI-модуля.

После передачи запроса обозревателя CGI-приложению сервер передает ему также данные из командной строки запроса. CGI-приложение формирует ответ и помещает его в выходной поток (на стандартном устройстве вывода), затем сервер посылает этот ответ с использованием протокола HTTP обратно обозревателю.

В случае параллельной обработки нескольких запросов сервер запускает отдельный процесс для каждого запроса, причем для каждого процесса создается копия модуля расширения в памяти компьютера, на котором находится Web-сервер. При большом размере исполняемого CGI-файла сильно увеличивается время отклика сервера на запрос обозревателя, поскольку потребуется время для загрузки модуля CGI с диска в память. Причем, если CGI-программа является интерпретируемой (например, написана на языке PHP или Perl), она будет выполняться еще медленнее, т. к. в этом случае, кроме загрузки модуля CGI, загружается интерпретатор PHP или Perl и производится интерпретация команд. При наличии сотен или тысяч обращений к серверу произойдет запуск сотен или тысяч CGI-программ, каждая из которых будет обрабатывать соответствующий запрос.

Отметим, что в некоторых операционных системах, в частности в UNIX, могут повторно использоваться уже загруженные программные коды модуля CGI первого процесса с созданием для каждого нового обращения только своего экземпляра данных.

Интерфейс *WinCGI* (протокол) отличается от интерфейса CGI тем, что управляющие параметры передаются через ini-файл, а входной и выходной поток данных перенаправлены в специальные файлы. Этот интерфейс является реализацией интерфейса CGI для операционной системы Windows 3.1.

Более перспективными интерфейсами для разработки дополнительных модулей расширения Web-сервера являются интерфейсы ISAPI/NSAPI. При использовании этих интерфейсов модули расширения реализуются в виде библиотек DLL. Такой механизм обеспечивает экономию ресурсов сервера и увеличение скорости обработки запросов.

Интерфейс ISAPI может применяться также для создания ISAPI-фильтров, которые, в отличие от модулей ISAPI, используются для контроля всего потока данных между сервером и обозревателем на уровне протокола HTTP. ISAPI-фильтры можно применять для динамической перекодировки, шифрования, сбора статистической информации о работе сервера.

Web-приложения с модулями расширения клиентской части

Для создания динамических эффектов при просмотре Web-страниц в модулях расширения клиентской части (активность на стороне клиента) используют апплеты, подключаемые программы, объекты ActiveX и сценарии. Архитектура Web-приложения с модулями расширения клиентской части показана на рис. 31.4.

Исполняемые программы являются первым поколением клиентских расширений. Для организации работы такой программы в Web-приложении необходимо ее предварительно установить и сконфигурировать обозреватель. Поэтому при использовании исполняемых программ очень трудно обеспечить совместимость Web-приложения с различными обозревателями. Кроме того, использование такой технологии нарушает принцип стандартности клиентского интерфейса.

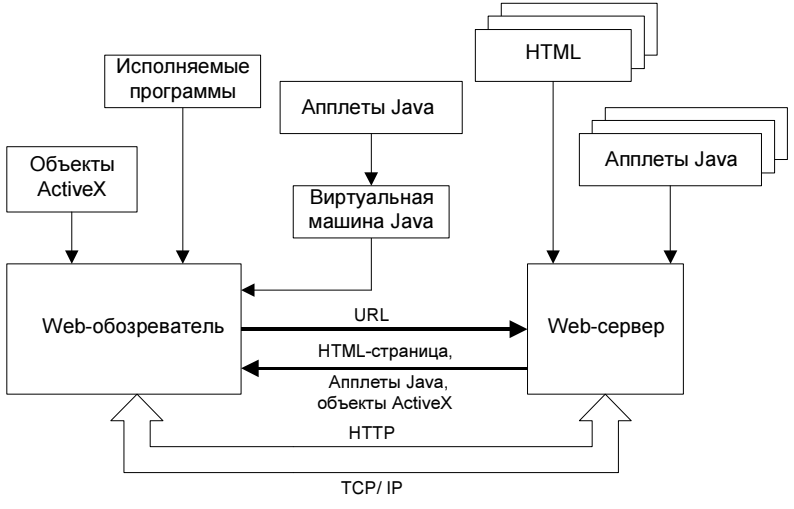


Рис. 31.4. Схема Web-приложения с модулями расширения обозревателя

Апплеты Java применяются для создания динамически формируемого интерфейса пользователя. Апплет представляет собой байтовый код, который интерпретируется виртуальной Java-машиной, входящей в состав обозревателя. Загрузка апплета производится при загрузке Web-документа. Принцип функционирования апплета делает возможным подключение классов Java, хранящихся на сервере. Код апплета может начать выполняться сразу после загрузки в компьютер клиента или активизироваться с помощью специальных команд.

Используя библиотеку классов Java, можно создавать мультимедийные страницы и организовывать распределенные процедуры обработки данных с использованием различных серверов и протоколов.

Апплет может быть специализирован для работы с внешними базами данных. С этой целью в Java включены наборы классов для поддержки графического пользовательского интерфейса и классы для доступа к БД.

Для включения дополнительного действия в Web-приложение достаточно включить тег апплета в Web-документ и поместить апплет-класс в библиотеку апплетов на сервере. При этом изменения в конфигурацию Web-сервера вносить не нужно.

Для взаимодействия апплета Java с внешним сервером баз данных разработан специализированный протокол JDBC (Java DataBase Connectivity — совместимость Java с базами данных), который построен на принципах интерфейса ODBC и применяется для стандартизации кода апплета Java при организации доступа к различным СУБД.

Сравним достоинства и недостатки использования технологии Java и наиболее распространенного в настоящее время интерфейса CGI. Технология апплетов Java является более гибкой. Апплет выполняется локально на машине пользователя, поэтому он может обеспечивать динамическое взаимодействие с пользователем гораздо быстрее. Кроме того, апплет может использоваться для выполнения функций, недоступных CGI-модулю. С точки зрения безопасности данных более целесообразно применять CGI-модули, т. к. они выполняются на стороне сервера и не могут получить доступ к ресурсам компьютера клиента.

Архитектура Web-приложений, публикующих БД

Для публикации БД на Web-страницах в архитектуру приложений вводятся дополнительные уровни, включающие сервер БД, сервер приложений и источник БД.

При такой архитектуре Web-сервер передает запрос на генерацию Web-страниц своей программе-расширению, которая на основе информации БД формирует требуемый документ. Затем Web-сервер отсылает готовые страницы обратно обозревателю. Как мы знаем, для формирования динамических страниц используются различные средства и технологии: ASP- и IDC/HTX-страницы, программы-расширения сервера на основе интерфейсов CGI и ISAPI.

При использовании ASP- и IDC/HTX-страниц запрос на получение динамически формируемой Web-страницы передается специальным динамическим библиотекам, входящим в состав Web-сервера. Например, если в качестве Web-сервера используется

Personal Web Server и публикация осуществляется средствами IDC/HTX, то применяется динамическая библиотека httpodbc.dll. Такие библиотеки анализируют файл ASP или IDC- и HTX-файлы, которые используются в качестве шаблонов.

Напомним, что публикация БД на Web-страницах бывает статической и динамической. Программы-расширения Web-сервера, публикующие базы данных, включают в себя элементы, характерные для обычных приложений БД. При статической или динамической публикации должна использоваться архитектура приложений БД, дополненная характерными компонентами архитектуры Web-приложений. Рассмотрим подробнее архитектуру Web-приложений, использующих БД.

Двухуровневые Web-приложения

При двухуровневой архитектуре Web-приложений источник БД хранится на том же компьютере, где находится Web-сервер. Для доступа к источнику БД используются модули расширения. В простейшем случае в архитектуру Web-приложений добавляется источник БД (рис. 31.5).

Функционирование Web-приложения при такой архитектуре заключается в следующем. Обозреватель для начала работы с Web-приложением отсылает URL-адрес главной страницы приложения Web-серверу. Последний, обработав запрос URL, высылает требуемую страницу в формате HTML обратно обозревателю. Эта страница несет общую информацию о Web-приложении и позволяет пользователю выбрать нужную ему функцию из предоставляемых приложением. Далее возможно несколько вариантов работы Web-приложения.

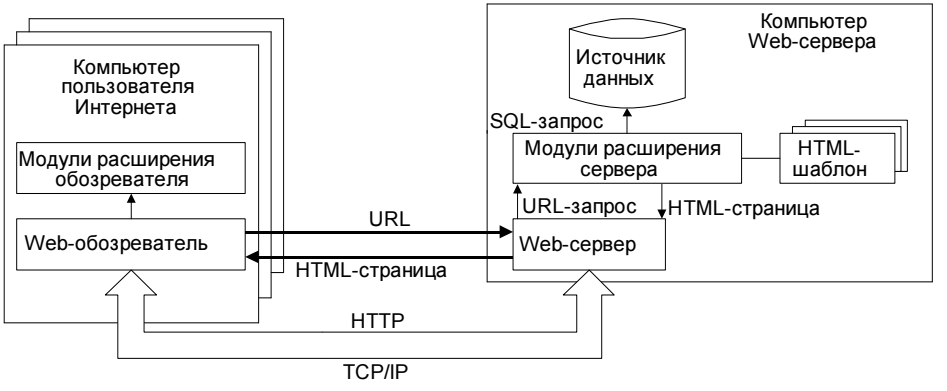


Рис. 31.5. Архитектура Web-приложения, использующего БД

Если пользователю нужна определенная информация из БД, то обозреватель по ссылке, находящейся в загруженной HTML-странице, формирует URL-запрос к модулю расширения сервера. Используемые при этом технологии различаются в зависимости от типа Web-сервера и других особенностей Web-приложения. Например, если на Web-узле установлен Web-сервер Microsoft Internet Information Server, то это может быть технология ASP- или IDC/HTX-страниц, интерфейсы CGI или ISAPI, а если установлен сервер Apache, то интерфейс CGI.

Если необходимо сформировать запрос URL с параметрами, то на уровне обозревателя могут использоваться сценарии JavaScript для проверки правильности ввода параметров запроса.

После того как пользователь выбрал ссылку, обозреватель отсылает URL Web-серверу. Для обработки запроса сервер вызывает требуемый модуль расширения и передает ему параметры URL. Модуль расширения сервера формирует SQL-запрос к БД.

Из модуля расширения сервера доступ к БД может осуществляться различными способами и на основе разных интерфейсов. Например, в случае использования технологии ASP-страниц применяются объектная модель ADO, объектный интерфейс OLE DB, интерфейс ODBC. Также возможен вариант непосредственного доступа к БД. Например, в случае модуля ISAPI, разработанного в среде Delphi, для доступа к БД может использоваться один посредник — драйвер BDE (Borland Database Engine), входящий в состав программных средств модуля расширения сервера.

Недостатки рассмотренной двухуровневой архитектуры:

- повышенная нагрузка на Web-сервер, связанная с тем, что все действия по обработке URL-запросов, извлечению информации из БД и формированию HTML-страниц выполняются Web-сервером и модулями расширения Web-сервера;
- низкий уровень безопасности из-за невозможности обеспечить требуемый уровень защиты информации в БД от сбоев во время обращения к базе данных из модуля расширения сервера или конфиденциальности информации БД от администратора Web-узла.

Для преодоления указанных недостатков применяются Web-приложения с большим числом уровней.

Трехуровневые Web-приложения

При включении в Web-приложение промежуточного уровня, основанного на технологии "клиент-сервер", его архитектура расширяется до трехуровневой. При такой архитектуре клиентский уровень занимает обозреватель, на уровне сервера находится сервер БД, а на промежуточном уровне размещаются Web-сервер и модули расширения сервера. Модуль расширения сервера выступает преобразователем протоколов между клиент-серверным приложением БД и Web-сервером (рис. 31.6).

Введение уровня Web-сервера в клиент-серверные приложения БД расширяет возможность их применения как межплатформенного приложения. Принципы взаимосвязи обозревателя и Web-сервера остаются теми же, что и в предыдущей архитектуре. Отличия этой архитектуры заключаются в организации взаимодействия модуля расширения сервера и источника БД.

Для получения данных модуль расширения Web-сервера формирует и отсылает SQL-запрос удаленному серверу БД. На компьютере, где установлен удаленный сервер БД, содержится и сама база данных. После получения SQL-запроса удаленный сервер направляет его SQL-серверу (серверу баз данных). SQL-сервер обеспечивает выполнение запроса и выдачу модулю расширения Web-сервера результатов запроса.

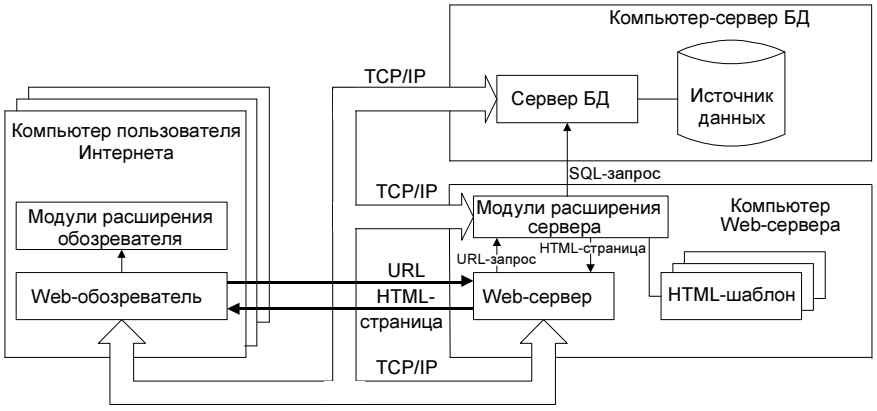


Рис. 31.6. Архитектура трехуровневого Web-приложения, использующего БД

Таким образом, в трехуровневой архитектуре вся обработка SQL-запроса выполняется на удаленном сервере. Достоинства такой архитектуры по сравнению с предыдущей состоят в следующем:

- уменьшение сетевого трафика — в сети циркулирует минимальный объем информации;
- увеличение уровня безопасности информации, поскольку обработка запросов к базе данных выполняется сервером БД, который запрещает одновременное изменение одной записи различными пользователями и реализует механизм транзакций;
- повышение устойчивости Web-приложения к сбоям;
- взаимозаменяемость компонентов архитектуры приложения;
- снижение сложности модулей расширения Web-сервера, в которых отсутствует программный код, связанный с контролем БД и разграничением доступа к ней.

Недостатком рассмотренной архитектуры является увеличение времени обработки запросов, связанное с дополнительным обращением по сети к серверу БД. Для устранения этого недостатка между сервером БД и Web-сервером должны использоваться высокоскоростные надежные линии связи.

Многоуровневые Web-приложения

Дальнейшее развитие архитектуры Web-приложений и технологии "клиент-сервер" привело к появлению многоуровневой архитектуры, в которой между модулем расширения Web-сервера и базой данных, кроме сервера БД, дополнительно вводится сервер приложений. *Сервер приложений* является промежуточным уровнем, который обеспечивает организацию взаимодействия клиентов ("тонких" клиентов) и сервера БД (рис. 31.7).

Напомним, что сервер приложений может использоваться для выполнения различных функций, которые в предыдущей архитектуре выполнялись сервером БД или модулем расширения Web-сервера.

В качестве "тонкого" клиента в этой архитектуре выступает программа-модуль расширения Web-сервера. Сервер приложений может обеспечивать взаимодействие с Web-

серверами и серверами БД, функционирующими на различных аппаратно-программных платформах (на компьютерах различных типов, под управлением различных операционных систем). Такая архитектура является основой для интрасетей, создаваемых на основе существующих локальных сетей.

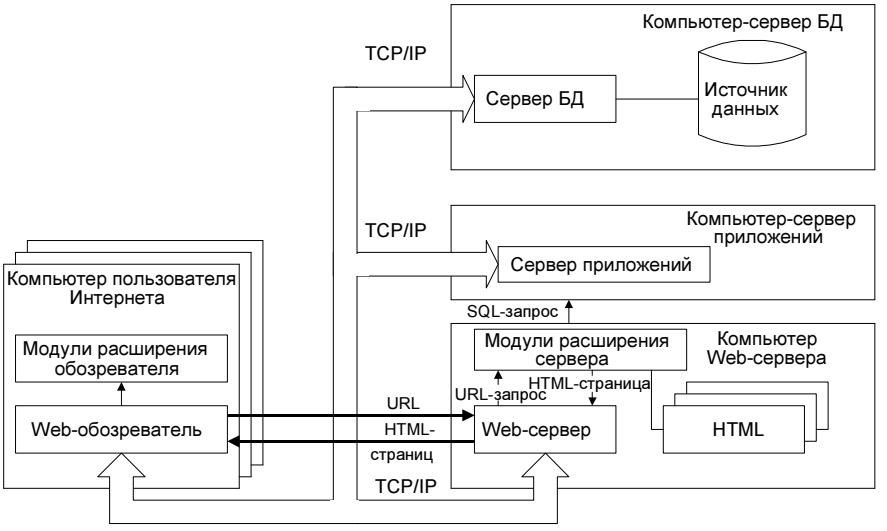


Рис. 31.7. Архитектура многоуровневого Web-приложения

Введение дополнительного уровня Web-сервера позволяет публиковать информацию из БД локальных сетей в Интернете, получать информацию от других интрасетей или Web-узлов. Кроме того, при частичной или полной реорганизации внутренней архитектуры локальных сетей появляется возможность использовать преимущества сетей интранет, касающиеся упрощения дополнительного подключения новых пользователей и администрирования локальной сети.

Отметим, что в некоторых архитектурах информационных систем Web-сервер может структурно объединяться с сервером приложений. В этом случае программные средства, входящие в состав модуля расширения, играют роль сервера приложений.

Основные достоинства многоуровневой архитектуры Web-приложений:

- разгрузка Web-сервера от выполнения части операций, перенесенных на сервер приложений, и уменьшение размера модулей расширения сервера вследствие их освобождения от лишнего кода;
- обеспечение более гибкого межплатформенного управления между Web-сервером и сервером БД;
- упрощение администрирования и настройки параметров сети — при внесении изменений в программное обеспечение или конфигурацию сервера БД не нужно вносить изменения в программное обеспечение Web-сервера.

При функционировании Web-приложений с использованием многоуровневой архитектуры сохраняется возможность параллельной работы Web-обозревателей и клиентских приложений БД (рис. 31.8).

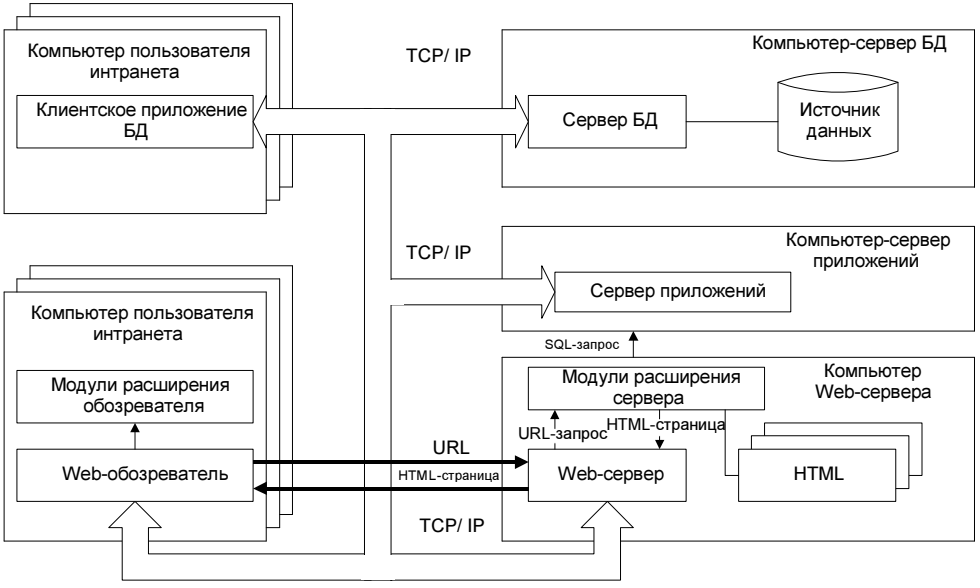


Рис. 31.8. Архитектура смешанного Web-приложения

В этом случае говорят о *смешанном* Web-приложении. При такой архитектуре Web-приложения и клиентские приложения БД могут параллельно получать доступ к источнику БД.

Web-приложения на основе CORBA

В настоящее время одним из перспективных направлений развития интрасетей является использование технологии CORBA (Common Object Request Broker Architecture — общая архитектура с брокером при запросе объекта) в сочетании с апплетами Java. CORBA представляет собой шину (интерфейс) распределенных объектов с открытыми стандартами, используемую в клиент-серверных системах. Единственной конкурентоспособной технологией, обладающей аналогичными возможностями, является технология DCOM (Distributed Component Object Model — распределенная объектная модель компонентов) фирмы Microsoft.

Стандарт CORBA не зависит от платформ и операционных систем. Поскольку технология CORBA хорошо интегрирована с Java, мы можем получить преимущества от совместного использования в трехуровневой архитектуре продуктов Delphi и JBuilder (интегрированной среды разработки на языке Java фирмы Borland).

Технология Java предполагает большую гибкость при разработке распределенных приложений, но не поддерживает в полной мере технологию "клиент-сервер". Интерфейс CORBA позволяет обеспечить связь переносимых приложений Java и объектов CORBA. Технология объектов CORBA предназначена для использования в Web-приложениях вместо CGI-интерфейса.

В результате объединения Java-апплетов и CORBA-интерфейса появилось новое понятие — *объектная модель Web*, означающее использование объектных моделей различ-

ных интерфейсов (модели CORBA, ADO и др.) при построении Web-приложений. Архитектура такого многоуровневого клиент-серверного Web-приложения, построенного на основе технологии CORBA и Java, приведена на рис. 31.9.

На первом уровне находится клиентское приложение — обозреватель. В нем выполняется клиентский апплет Java, из которого может осуществляться обращение к объектам CORBA.

На втором уровне находится Web-сервер, обрабатывающий HTTP-запросы и CORBA-вызовы клиентских приложений.

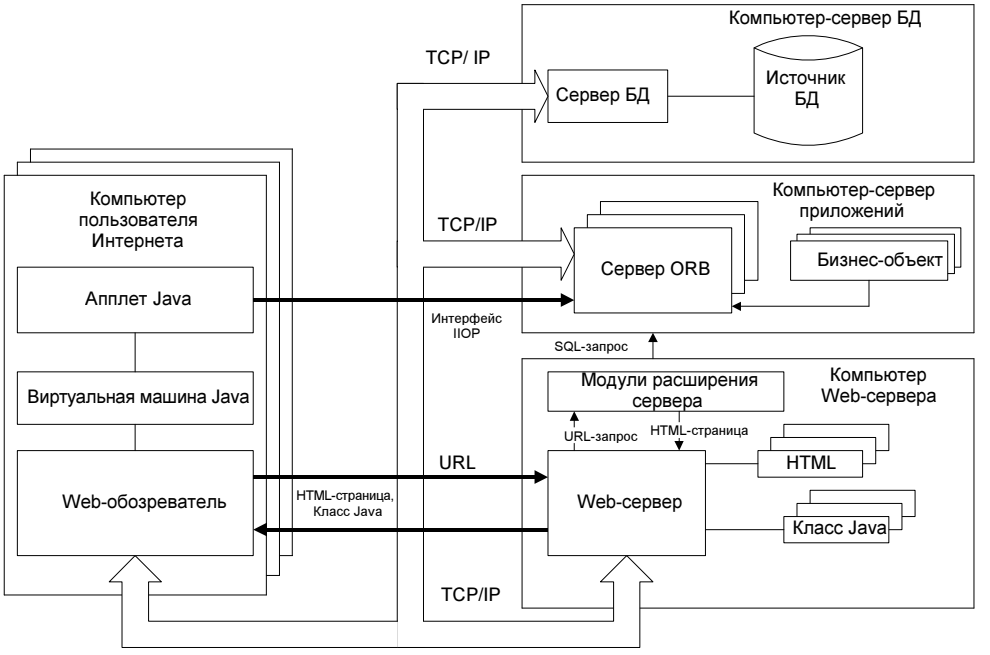


Рис. 31.9. Архитектура многоуровневого Web-приложения на основе технологии CORBA

Третий уровень — это уровень сервера приложений. В его роли могут выступать серверы ORB (Object Request Broker — посредник запросов объектов) или распределенные объекты CORBA, функционирующие как серверы приложений промежуточного звена и выполняющие прикладные функции и набор компонентных сервисов (услуг). Серверы ORB являются унифицированными фрагментами программы, используемыми в распределенных приложениях в качестве связующего звена между клиентскими приложениями и сервером.

Объекты CORBA взаимодействуют с серверами БД последнего уровня, используя, например, SQL в случае реляционных БД. Кроме того, объекты CORBA на сервере могут взаимодействовать и друг с другом. В основе механизма взаимодействия между объектами CORBA лежит протокол IIOP (Internet Inter-ORB Protocol — интернет-протокол взаимодействия ORB). Протокол IIOP основан на протоколе TCP/IP с добавленными компонентами обмена сообщениями и функционирует как общий опорный протокол при организации взаимодействия серверов ORB и объектов CORBA. В дополнение к IIOP, в технологии CORBA используются ESIOP-протоколы (Environment-Specific

Inter-ORB Protocols — зависящие от среды протокола взаимодействия ORB), применяемые в специализированных сетевых средах.

Java-клиент может непосредственно взаимодействовать с объектом CORBA, используя Java ORB. При этом серверы CORBA замещают уровень HTTP-сервера и выступают в качестве программного обеспечения промежуточного уровня, обеспечивая взаимодействие между объектами (object-to-object). Интерфейс CORBA IIOP функционирует в Интернете так же, как и протокол HTTP.

Протокол HTTP в этом случае используется для загрузки Web-документов, апплетов и графики, а CORBA — для организации клиент-серверных приложений с помощью апплетов Java.

Серверный компонент CORBA предоставляет "настраиваемый" интерфейс, который можно конфигурировать с помощью визуальных средств. Объект CORBA обладает определенными функциональными возможностями, реализует инкапсуляцию свойств и методов, генерируемых объектами событий. Можно создавать ансамбли объектов, "стыкуя" выходные события с входными методами. Разработка таких визуальных объектов поддерживается средствами быстрой разработки приложений (RAD). В частности, объекты CORBA поддерживаются в Delphi.

На последнем, четвертом, уровне размещается сервер баз данных или другой источник данных, т. е. практически любой источник информации, к которому CORBA может получить доступ. Сюда входят мониторы процедур транзакций (TP Monitors), MOM (Message-Oriented Middleware — промежуточное программное обеспечение, ориентированное на обмен сообщениями), ODBMS (объектные СУБД), электронная почта и т. д.

Таким образом, CORBA обеспечивает инфраструктуру распределенных объектов, что позволяет приложениям распространяться через сети, языки, границы компонентов и операционные системы. В свою очередь, Java обеспечивает инфраструктуру переносимых объектов, которые работают на всех основных операционных системах. То есть CORBA дает независимость от сетей, а Java — независимость от реализации.

В среде Delphi 7 для поддержки технологии CORBA на вкладке **Corba** диалогового окна **New Items** Хранилища объектов, открываемого командой **File | New | Other**, имеются два пункта: **CORBA Client Application** и **CORBA Server Application**. С их помощью открываются диалоговые окна **IDL2Pas Create Client Dialog** и **IDL2Pas Create Server Dialog**, облегчающие генерирование клиентской и серверной части приложений CORBA на основе использования файлов IDL.

В названных диалоговых окнах для клиента и для сервера CORBA можно выбрать создание двух видов приложений: консольных приложений и приложений Windows. Файлы IDL (Interface Definition Language) служат для описания интерфейсов объектов или подпрограмм, используемых в распределенных приложениях CORBA.

Для организации связи программных расширений Web-сервера с БД используются современные интерфейсы доступа к данным: OLE DB, ADO и ODBC. Эти интерфейсы являются промежуточным уровнем между источником данных и приложением, в качестве которого выступают программные расширения Web-сервера. Рассмотрим особенности архитектуры Web-приложений, использующих интерфейсы доступа к данным OLE DB, ADO и ODBC.

Web-приложения на основе OLE DB, ADO и ODBC

Современные интерфейсы OLE DB, ADO и ODBC, внедряемые фирмой Microsoft, позволяют осуществлять доступ к различным источникам данных единообразными способами. Доступ к данным основан на интерфейсе OLE DB, позволяющем связывать и внедрять объекты из любых источников. (Напомним, что интерфейс OLE DB является универсальной технологией доступа к данным через стандартный интерфейс COM и основан на механизме сервис-провайдеров (поставщиков услуг).)

Архитектура Web-приложений, использующих интерфейсы OLE DB, ADO и ODBC, приведена на рис. 31.10.

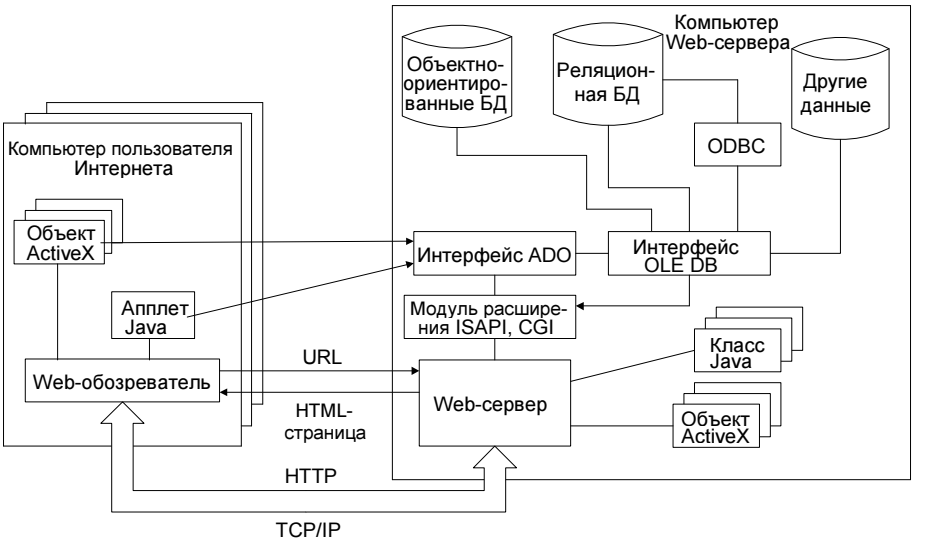


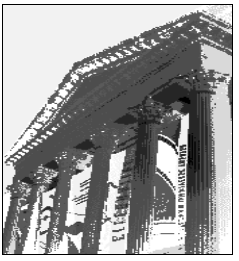
Рис. 31.10. Архитектура Web-приложений с интерфейсами OLE DB, ADO и ODBC

Интерфейс ADO представляет собой еще более высокий уровень абстракции, чем интерфейс OLE DB. Он реализован в виде иерархической модели объектов для доступа к различным OLE DB-провайдерам данных. В модель ADO входит набор объектов, которые обеспечивают соединение с провайдером данных, создание SQL-запроса к данным, создание набора записей на основе запроса и др.

Особенность функционирования Web-приложений, использующих интерфейс ADO, заключается в том, что обозреватель может извлекать информацию из любого источника данных, находящегося в Интернете, заранее не имея представления о логической структуре, типе и физическом формате источника данных. То есть появляется возможность публиковать требуемую информацию в Интернете, не показывая внутреннюю структуру данных.

Таким образом, интерфейс ADO позволяет стандартизировать существующие интерфейсы для доступа к данным в сетях Интернет/интранет. Он объединяет все существующие интерфейсы и предоставляет единообразный способ доступа к любому источнику данных через любой интерфейс.

ГЛАВА 32



Web-серверы и интерфейсы

В этой главе рассматриваются характеристики популярных Web-серверов: Microsoft Internet Information Server, Apache, Netscape Enterprise, а также описываются общепринятые интерфейсы программирования Web-приложений.

Обзор Web-серверов

Дадим краткую характеристику наиболее распространенным Web-серверам, используемым в корпоративных сетях и в "домашних" компьютерах, которые могут применяться для публикации информации из БД. Для начала определим само понятие Web-сервера. *Web-сервер* — это программное средство, установленное на Web-узле глобальной или корпоративной сети и позволяющее пользователям сети получать доступ к гипертекстовым документам этого Web-узла. Иногда под Web-сервером понимают программное обеспечение Web-сервера вместе с его аппаратной частью, т. е. с компьютером, на котором Web-сервер установлен.

В общем случае программное обеспечение Web-сервера может устанавливаться на компьютеры общего назначения, предназначенные для решения различных задач, не обязательно связанных с технологиями Интернета. Поэтому более корректно использовать понятие Web-сервера для обозначения только программного обеспечения, а компьютер с сетевой структурой и операционную систему называть средой работы Web-сервера, или *платформой*.

Отметим, что Web-серверы используются для следующих целей:

- создание корпоративных сетей интранет на основе принципов Интернета, многоуровневой архитектуры и клиент-серверных технологий;
- подключение корпоративных сетей интранет к Интернету для доступа к предоставляемым в нем услугам;
- публикация информации из корпоративных сетей интранет, в том числе содержимого БД из информационных систем, функционирующих в среде интранет;
- распространение собственной информации, находящейся на домашнем компьютере, создание собственного сайта.

В настоящее время в Интернете функционирует большое число различных серверов, используемых для обеспечения различных функций. Кроме того, есть многочисленные однотипные серверы, разработанные различными производителями.

В Интернете информацию о серверах можно найти на ряде сайтов, содержащих ежемесячные обзоры всего, что касается данной темы. Например, один из наиболее популярных источников информации о статистике использования Web-серверов — узел **http://www.netcraft.com/survey** компании Netcraft, который кроме информации о серверах и платформах содержит список адресов, по которым можно найти сведения об основных Web-серверах.

На выбор сервера большое влияние оказывает платформа, на которой должен работать Web-сервер. Анализ различных источников показывает, что Web-узлы можно организовывать на компьютерах любых типов, имеющих достаточные ресурсы и производительность. Активно используемых типов операционных систем намного меньше, чем типов компьютеров. Статистика показывает, что для высокопроизводительных объемных узлов наиболее часто используется операционная система UNIX (около 80% Web-серверов работают под ее управлением), а для средне- и низкопроизводительных узлов чаще всего используется Windows NT/2000 (меньше 20% Web-серверов).

Операционные системы для Web-серверов

В Интернете есть несколько основных операционных систем. Как сказано ранее, самыми распространенными программными платформами для Web-серверов являются UNIX-подобные операционные системы.

Операционная система UNIX и ее клоны получили наибольшее распространение в среде Интернета по следующим причинам:

- ОС UNIX может работать на значительно большем количестве аппаратных платформ по сравнению с другими операционными системами. Она распространяется в исходных кодах, поэтому легко может быть перекомпилирована для любой аппаратной платформы;
- ОС UNIX раньше других начала применяться в Интернете;
- ОС UNIX поддерживает большое количество услуг, ориентирована на работу с бóльшим количеством процессоров, а также адресов IP;
- ОС UNIX устойчиво работает в условиях даже весьма загруженных сетей.

Естественно, за все надо платить, и в результате система UNIX является самой сложной для изучения и конфигурирования из всех существующих операционных систем.

Операционные системы Windows 2000 Server и Windows NT Server широко распространены, часто используются в качестве платформы Web-серверов и являются единственными реальными конкурентами для ОС UNIX. Отметим, что Windows 2000 Server представляет собой доработанный вариант Windows NT Server, включающий в себя достоинства Windows 98. Основное преимущество Windows 2000/NT Server заключается в легкости настройки и освоения работы. Поэтому для начинающих пользователей лучше начать работу в Web с одной из этих операционных систем.

Для Web-узла с малой или средней нагрузкой операционная система Windows 2000/NT Server фирмы Microsoft работает достаточно надежно. Об этом свидетельствует тенденция роста в процентном соотношении числа Web-серверов, использующих Windows 2000/NT Server. Отметим, что другие операционные системы Windows 9*x* фирмы Microsoft не зарекомендовали себя как надежные платформы для Web-узла.

В Windows 2000 расширены средства поддержки взаимодействия с другими операционными системами. Так, Windows 2000 позволяет организовывать взаимодействие с Windows NT Server 3.51 и 4.0, поддерживает клиентов с операционными системами Windows 95, Windows 98 и Windows NT Workstation 4.0, с большими и средними ЭВМ с помощью шлюзов транзакций и очередей. Файловый сервер для Macintosh позволяет клиентам Macintosh организовывать общий доступ к файлам и использовать общие ресурсы Windows 2000/NT Server.

Тем не менее, ОС Windows 2000/NT Server не обеспечивает требуемую гибкость при администрировании и расширении возможностей Web-узла.

Отметим, что тенденция роста использования Windows 2000/NT будет сохраняться за счет вхождения в Интернет большого количества пользователей Intel-компьютеров. Windows-ориентированные серверы могут устанавливаться и настраиваться автоматизированно, в отличие от UNIX-ориентированных Web-серверов, которые настраиваются только вручную.

Операционная система Mac OS X используется пользователями Macintosh, работающими в Web. В настоящее время эта операционная система значительно уступает по популярности Windows NT и UNIX ввиду того, что имеет ограниченные возможности по взаимодействию с динамическими узлами и не приспособлена к режиму повышенной нагрузки. Mac OS X не является лучшим выбором операционной системы для Web-сервера, но остается популярной среди пользователей Macintosh.

Рассмотрим теперь наиболее популярные типы Web-серверов.

Сервер Apache

Серверы Apache довольно хорошо отлажены и протестированы разработчиками и пользователями. Группа разработчиков Apache придерживается строгих стандартов в отношении выпуска новых версий. При обнаружении ошибок в работе сервера компания Apache Development Group выпускает корректирующие файлы или новые версии продукта. Эта компания является международной организацией добровольцев, разработавших данный программный продукт для некоммерческого распространения среди широкого круга пользователей. Созданный под покровительством компании Apache Digital Corporation, проект Web-сервера Apache развивался как ветвь проекта NCSA httpd — одного из самых первых и наиболее эффективных из существующих серверов Интернета.

Само слово Apache звучит похоже на "a patchy" ("лоскутный"). Одновременно это — просто красивое название, связанное с американским индейским племенем Apache, известным своим военным мастерством и неутомимостью.

По сравнению с другими серверами Apache показал себя более устойчивым, более быстрым, имеющим более широкий набор функций и возможностей. Кроме того, Web-

сервер Apache характеризуется открытой архитектурой, заключающейся в том, что этот сервер распространяется в исходных кодах и позволяет легко наращивать дополнительные возможности.

Его безусловное лидерство в Интернете объясняется тем, что сервер был разработан для самой популярной платформы UNIX. Хотя сервер Apache распространяется бесплатно, организация, разработавшая и обслуживающая этот мощный пакет (см. сервер **www.apache.org**), обеспечивает свое функционирование с помощью пользователей, которые спонсируют его развитие и сопровождение.

Используя открытый код Apache, разработчик может создавать собственные конфигурации сервера, компилируя внесенные в код изменения. Apache имеет модульную структуру, т. е. в его состав входит набор модулей, которые предназначены для обеспечения требуемых функций сервера и могут быть динамически включены в конфигурацию даже во время активной работы сервера. Сервер Apache позволяет использовать CGI-сценарии, написанные на Perl или PHP.

Перечислим основные особенности функционирования сервера Apache:

- является мощным, гибким, HTTP/1.1-совместимым сервером;
- поддерживает современные протоколы;
- имеет легко перестраиваемую конфигурацию с возможностью установки дополнительных функций (модулей) от сторонних производителей;
- может быть сконфигурирован с использованием модулей API;
- снабжается полным исходным текстом и поступает с бесплатной лицензией на использование без ограничений;
- работает под управлением популярных операционных систем Windows NT/9*x*, NetWare 5.*x*, OS/2 и большинства версий UNIX;
- поддерживает ведение отчетной документации об ошибках и файлы коррекции.

Сервер Apache поддерживает следующие функции:

- доступ к базам данных, используемым для аутентификации, т. е. возможность установки защищенных паролем страниц с огромным количеством уполномоченных пользователей без перегрузки сервера;
- настройка реакции сервера на ошибки и сбои, заключающаяся в возможности устанавливать файлы или даже сценарии CGI, используемые сервером при возникновении ошибки (например, установка сценариев, позволяющих обрабатывать около 500 ошибок сервера, вести непрерывную диагностику и устранять неполадки по желанию пользователя);
- автоматическая обработка HTML-данных с изменяющейся структурой и модификация их для удобного представления информации клиенту;
- поддержка виртуальных хостов, заключающаяся в возможности настройки нескольких "домашних хостов", что позволяет серверу различать запросы, сделанные по различным IP-адресам; Apache также предоставляет возможность динамически настраивать функции "главного" виртуального хоста;
- генерация информации о настройках в удобном для пользователя формате;

- формирование в большинстве архитектур UNIX Apache так называемых журналов учета (log-файлов).

Отметим, что для сервера Apache отсутствуют официальное техническое обслуживание и поддержка. Тем не менее, для этой программы можно найти большое количество информации или советов. Например, список известных сбоев можно найти на Web-узле, а со службой поддержки от независимых разработчиков можно связаться через список рассылки Apache **comp.infosystems.www.servers.unix**, через коммерческие службы, например, Cygnus (**http://www.cygnus.com/product/idk/apache**) или ежемесячный дайджест от разработчиков Apache. Еще одним неплохим источником технической информации является Apache Week.

В перспективе для сервера Apache планируется разработать графический пользовательский интерфейс и окончательно адаптировать версию этого сервера для работы на платформе Windows NT/9*x*. Кроме того, основные направления политики внедрения сервера Apache в Интернете остаются прежними, а именно открытость архитектуры, тесная обратная связь с пользователями и поддержка последних версий протокола HTTP.

Сервер Microsoft Internet Information Server

Ближайшим конкурентом Apache является Microsoft Internet Information Server (MS IIS), входящий в состав системы Windows 2000/NT Server. Согласно опросу, проводимому компанией Netcraft, под управлением MIIS работает примерно в три раза меньше серверов по сравнению с управляемым Apache, т. е. этому серверу отдают предпочтение около 20% пользователей (Apache используют около 60% пользователей). Он обладает многими новыми функциональными возможностями, среди которых создание нескольких Web-узлов, новые средства администрирования, работа в режиме сервера новостей.

Еще одной особенностью стали определяемые пользователем специальные группы Интернета, где можно размещать различные ресурсы, например, принтеры, с целью упрощенного обращения к ним и просмотра.

Службы IIS позволяют использовать масштабируемые Web-приложения, а также публиковать данные из информационных систем в Интернете. Службы IIS поддерживают страницы ASP, которые являются средой создания серверных сценариев для разработки динамических интерактивных приложений Web-серверов. Они позволяют разработчикам объединять нужным образом страницы в формате HTML, команды сценариев и компоненты COM для создания мощных и гибких Web-приложений.

Упрощенной версией IIS является Microsoft Personal Web Server (PWS), предназначенный для работы в качестве настольного Web-сервера для распространения в сети информации с домашнего компьютера.

К недостаткам IIS относят то, что этот сервер разработан в основном для платформ Windows 2000/NT и 9*x*. Используется сервер, как правило, с операционной системой Windows 2000/NT, т. к. эта система гораздо более стабильна в работе и устойчива к сбоям по сравнению с операционными системами Windows 9*x*. Анализ статистических данных и отзывов пользователей показывает, что общая производительность и надежность

IIS на платформе Windows 2000/NT Server близка к среднему показателю для сервера Apache.

MS IIS 4.0 является встроенным в Windows NT 5.0 сервером и поставляется бесплатно как часть пакета NT Server.

Microsoft Internet Information Server позволяет так же расширять возможности сервера, как и сервер Apache. Используя интерфейс COM, можно подключать различные языки создания сценариев для этого Web-сервера. Но в данном случае интерпретаторы сценариев не входят непосредственно в состав сервера, что несколько замедляет обработку сценариев.

Наибольшее различие между Apache и IIS заключается в возможности изменять конфигурацию сервера Apache без остановки его работы, изменяя в текстовом режиме файлы настройки, что позволяет динамически включать и выключать необходимые модули. Однако графический интерфейс IIS развит гораздо лучше, в то время как аналогичный модуль Apache все еще в стадии разработки. Это объясняется тем, что первоначально Apache разрабатывался для UNIX-подобных систем с интерфейсом в виде командной строки.

Среди недостатков IIS можно выделить недостаточный уровень безопасности дисков с файловой системой NTFS, в результате которой возникает зависимость от сети и системы безопасности Windows NT.

Положительным фактом для IIS является возможность получения технической поддержки этого сервера. В интерактивном режиме предоставляется вся информация, касающаяся установки и обслуживания. Дополнительно в пакете MS IIS имеются отдельные файлы справки по утилитам.

Сервер Netscape Enterprise

Третье место по рейтингу использования серверов занимает сервер компании Netscape, которая осуществляет продажу разнообразных серверных продуктов (Communications, Commerce и Enterprise). Сервер Netscape Enterprise используют около 7% Web-узлов. Этот сервер уверенно конкурирует с основными типами серверов.

Достоинством этого сервера является то, что фирма Netscape разработала версии сервера, которые работают в системах Windows 95, NT и UNIX.

Сервер Netscape характеризуется простотой в использовании и надежностью в работе, "интегрированностью" высокопроизводительных функций в интерфейс пользователя.

Информацию по программным продуктам Netscape можно найти по адресу **www.netscape.com**.

Интерфейсы Web-приложений

В следующих разделах описаны общепринятые интерфейсы программирования Web-приложений — общий интерфейс взаимодействия CGI и интерфейс программирования серверных приложений ISAPI.

Общий интерфейс взаимодействия CGI

CGI-модуль представляет собой консольный исполняемый файл, загружаемый сервером для обработки данных обозревателя и генерации HTML-документа. Обмен данными между сервером и CGI-модулем происходит через стандартные входной и выходной потоки. Кроме того, для передачи параметров и получения системной информации могут задействоваться переменные окружения. При использовании CGI-интерфейса (протокола) можно формировать HTML-документ динамически (в ответ на запрос пользователя из обозревателя).

CGI-модуль может вызываться из обозревателя для динамического формирования HTML-документов на основании данных, введенных посетителями Web-сервера при помощи форм или путем выбора ими ссылок. Напомним, что обработку данных, введенных пользователем в интерфейсные элементы формы, можно выполнить с помощью обработчика, URL которого задается атрибутом `ACTION` тега формы, а способ передачи данных из формы на Web-сервер определяет атрибут `METHOD` тега формы.

При разработке CGI-модуля необходимо учитывать, каким методом HTTP-протокола будут передаваться данные между обозревателем и сервером. Основными методами передачи параметров запроса являются `POST` и `GET`. Метод `POST` применяется для посылки информации, включенной в HTTP-запрос и относящейся к ресурсу, указанному URL-адресом CGI-модуля. Метод `GET` используется для передачи информации в URL-адресе CGI-модуля. В зависимости от метода, примененного в HTTP-запросе, сервер формирует передаваемые для CGI-модуля параметры различными способами. В случае метода `GET` для получения параметров запроса привлекаются переменные окружения (переменные операционной системы MS-DOS, устанавливаемые командой `SET`), если же задействован метод `POST`, данные доставляются через стандартный поток ввода (рис. 32.1).

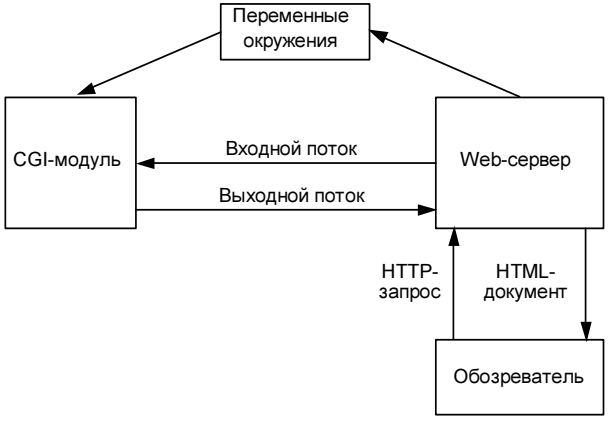


Рис. 32.1. Схема взаимодействия сервера и CGI-модуля

Например, см. листинг 32.1.

Листинг 32.1. Пример CGI-модуля

```
<HTML><HEAD><TITLE>Пример CGI-модуля</TITLE>
</HEAD>
```

```
<BODY>
<A HREF="http://localhost/scripts/my_cgi.exe?TEST=modul">
Загрузить CGI-модуль по запросу
http://localhost/Scripts/my_cgi.exe?TEST=modul</A>
<FORM ACTION=http://localhost/scripts/my_cgi.exe METHOD=POST>
<B>Введите ваше имя:</B>
<INPUT MAXLENGTH=60 NAME=NAME SIZE=40 VALUE="My_login">
<BR>Тип действия: <INPUT CHECKED NAME=TYPE TYPE=radio
                          VALUE=Registry>- Зарегистрировать
<INPUT NAME=TYPE TYPE=radio VALUE=Execute>Выполнить сценарий
<BR><INPUT NAME=Submit TYPE=Submit VALUE=Послать запрос>
<BR><INPUT NAME=Reset TYPE=Reset VALUE=Сброс><BR>
</FORM></P></BODY></HTML>
```

В окне обозревателя будут отображены ссылка (`Загрузить CGI-модуль по запросу http://localhost/Scripts/my_cgi.exe?TEST=modul`) и элементы управления, входящие в состав формы. При нажатии специальной кнопки **Послать запрос**, имеющей тип `Submit`, обозреватель отправляет запрос Web-серверу. Последний, в свою очередь, запускает CGI-модуль, находящийся по URL-адресу **http://localhost/Scripts/my_cgi.exe**. При этом данные, введенные пользователем в интерфейсные элементы формы, переносятся через стандартный поток ввода.

При выборе пользователем ссылки `Загрузить CGI-модуль по запросу http://localhost/Scripts/my_cgi.exe?TEST=modul` обозреватель передает запрос Web-серверу на загрузку CGI-модуля, расположенного по URL-адресу **http://localhost/Scripts/my_cgi.exe**. При этом для посылки запроса используется метод `GET`, и сервер отправляет CGI-модулю только данные, находящиеся в командной строке (`TEST=modul`) через переменную окружения `QUERY_STRING`.

Метод `GET` может использоваться и для передачи данных формы. В этом случае в теге формы указывается `METOD=GET`. При использовании метода `GET` данные размещаются в переменной окружения `QUERY_STRING` в следующем формате:

```
"Имя1=Значение1&Имя2=Значение2&Имя3=Значение3"
```

Здесь `Имя1`, `Имя2`, `Имя3` — значения параметров `NAME`, задающих имена полей формы. При этом количество параметров определяется исходя из числа элементов формы (в данном случае таких три). На место значений параметров подставляются данные из соответствующих полей.

Отметим, что не все значения интерфейсных элементов формы передаются на сервер. Например, значение служебного элемента типа `reset`

```
<INPUT NAME=Reset TYPE=reset VALUE=Сброс>
```

не передается на сервер, т. к. оно не несет полезной информации.

Если в приведенном выше примере HTML-документа параметр `METHOD` сопоставить с методом `GET`, то переменная `QUERY_STRING` примет значение

```
NAME=My_login&TYPE=Registry&Submit=Send
```

Это значение будет закодировано, т. е. символы пробела будут замещены символами `+`, а специальные и управляющие коды (в случае их использования) будут заменены на

последовательности символов вида `Xxx`, где `xx` — шестнадцатеричное представление символа.

Если в имена или в поля интерфейсных элементов входят символы, набранные в альтернативной кодировке, то при их передаче сервер будет также подставлять вместо каждого такого символа шестнадцатеричный код, предваряемый символом `%`.

Если в приведенном выше примере первый интерфейсный элемент оформить как `<INPUT MAXLENGTH=60 NAME=ИМЯ SIZE=40 VALUE="МОЕ ИМЯ">`, то переменная `QUERY_STRING` будет иметь следующее значение

```
%C8%CC%DF=%CC%CE%C5+%C8%CC%DF&TYPE=Registry&Submit=Send
```

В результате CGI-модуль генерирует HTML-документ, который помещается в стандартный поток вывода. Web-сервер задействует для управления и передачи различной информации CGI-модулю многочисленные переменные окружения, особенности использования которых мы рассмотрим в следующих разделах, включая стандартные потоки ввода/вывода.

Переменные окружения

Переменные окружения применяются в CGI-модуле для получения от Web-сервера служебной информации о самой программе Web-сервера, параметрах HTTP-запроса и другой информации, передаваемой сервером модулю. Например, в случае применения метода `GET` при формировании HTTP-запроса в переменную `QUERY_STRING` помещаются передаваемые пользовательские данные, а при альтернативном подходе (метод `POST`) в переменных окружения `CONTENT_TYPE` и `CONTENT_LENGTH` содержатся тип и длина передаваемой информации соответственно. Сами данные в последнем случае доставляются через стандартный входной поток.

Переменные окружения могут включать следующие данные:

- `SERVER_NAME` — символическое имя или IP-адрес компьютера, на котором запущен Web-сервер (задается в URL при обращении к этому Web-серверу);
- `SERVER_SOFTWARE` — название и версия Web-сервера, разделенные символом `/`;
- `GATEWAY_INERFACE` — версия CGI-интерфейса;
- `SERVER_PROTOCOL` — наименование и версия протокола передачи данных, используемого сервером, разделенные символом `/`;
- `SERVER_PORT` — номер порта, на который обозреватель посылает запросы Web-серверу;
- `REQUEST_METHOD` — метод HTTP-запроса;
- `CONTENT_LENGTH` — количество символов в стандартном входном потоке;
- `CONTENT_TYPE` — тип данных, находящихся в стандартном входном потоке;
- `SCRIPT_NAME` — виртуальный путь к исполняемому CGI-модулю, используемый для получения URL в CGI-модуле;
- `PATH_INFO` — полученный от клиента виртуальный путь к CGI-модулю;
- `PATH_TRANSLATED` — физический путь до CGI-модуля, преобразованный из значения `PATH_INFO`;

- QUERY_STRING — строка символов, следующая за знаком ? в URL данного запроса;
- REMOTE_HOST — символическое имя удаленной машины, с которой произведен запрос;
- REMOTE_ADDRESS — IP-адрес клиента;
- AUTH_TYPE — метод аутентификации (подтверждения подлинности), если Web-сервер поддерживает аутентификацию пользователей и CGI-модуль защищен от постороннего доступа;
- REMOTE_USER — имя пользователя в случае аутентификации;
- REMOTE_IDENT — имя пользователя, полученное от сервера (если сервер поддерживает аутентификацию);
- HTTP_ACCEPT — список типов MIME, известных клиенту и отделенных друг от друга запятой (тип/подтип, тип/подтип и т. д.);
- HTTP_USER_AGENT — название обозревателя, пославшего запрос;
- HTTP_REFER — URL документа HTML, из которого осуществляется вызов CGI-модуля;
- HTTP_ACCEPT — типы данных MIME, которые могут быть приняты обозревателем от Web-сервера;
- HTTP_ACCEPT_LANGUAGE — идентификатор национального языка для получения ответа от Web-сервера;
- HTTP_UA_PIXELS — разрешение видеоадаптера, установленное в компьютере пользователя;
- HTTP_UA_COLOR — допустимое число цветов в системе пользователя;
- HTTP_UA_CPU — тип центрального процессора в компьютере пользователя;
- HTTP_UA_OS — операционная система, под управлением которой работает обозреватель;
- HTTP_CONNECTION — тип соединения;
- HTTP_HOST — имя узла, на котором работает Web-сервер;
- HTTP_ACCEPT_ENCODING — тип схемы кодирования, используемой обозревателем для формирования запроса Web-серверу;
- HTTP_FROM — имя пользователя, установленное в настройках обозревателя;
- HTTP_PRAGHA — специальные команды Web-серверу;
- HTTP_AUTHORIZATION — информация для аутентификации обозревателя на Web-сервере.

Замечание

Тип MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения почтового стандарта Интернета) определяет протокол передачи почтовых сообщений, используемый взамен стандартного.

Для Web-сервера Microsoft PWS 95/2.0 в среде Windows 98 и адреса

```
HREF="http://igin/scripts/my_cgi.exe?TEST=modul"
```

задаваемого в адресной строке обозревателя, переменные окружения будут иметь следующие значения:

```
SERVER_SOFTWARE = Microsoft-PWS-95/2.0
SERVER_NAME = igin
GATEWAY_INTERFACE = CGI/1.1
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 80
REQUEST_METHOD = POST
HTTP_ACCEPT = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
PATH_TRANSLATED = C:\WebShare\wwwroot\
SCRIPT_NAME = /scripts/my_cgi.exe
QUERY_STRING = TEST=modul
REMOTE_HOST = 127.0.0.1
REMOTE_ADDR = 127.0.0.1
CONTENT_TYPE = application/x-www-form-urlencoded
HTTP_ACCEPT = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
HTTP_USER_AGENT = Mozilla/4.7 [en] (Win98; I)
HTTP_ACCEPT_LANGUAGE = en,ru
HTTP_CONNECTION = Keep-Alive
HTTP_HOST = igin
HTTP_ACCEPT_ENCODING = gzip
```

Остальные переменные принимают нулевое значение (`Null`).

Здесь: `igin` — имя компьютера, на котором выполняется CGI-модуль; `C:\WebShare\wwwroot\` — каталог, в котором находится CGI-модуль; `my_cgi.exe` — имя файла, содержащего модуль.

Для Web-сервера Apache/1.3.14 и адреса

```
http://igin/cgi-bin/my_cgi.exe
```

следующие переменные окружения могут иметь значения, отличные от приведенных ранее:

```
SERVER_SOFTWARE = Apache/1.3.14 (Win32)
SERVER_NAME = 127.0.0.1
SERVER_NAME = 127.0.0.1
REMOTE_HOST = (null)
HTTP_HOST = igin
PATH_TRANSLATED = d:\program\apache\htdocs\
SCRIPT_NAME = /cgi-bin/my_cgi.exe
```

Для Web-сервера MS IIS 5.0 в среде Windows 2000 Server и адреса, задаваемого в адресной строке обозревателя:

```
http://igin/Scripts/my_cgi.exe
```

следующие переменные окружения могут иметь значения, отличные от представленных выше:

```
SERVER_SOFTWARE = Microsoft-IIS/5.0
SERVER_NAME = igin
```

```
GATEWAY_INTERFACE = CGI/1.1
SERVER_PROTOCOL = HTTP/1.1
PATH_TRANSLATED = c:\inetpub\wwwroot
SCRIPT_NAME = /Scripts/my_cgi.exe
REMOTE_HOST = 127.0.0.1
REMOTE_ADDR = 127.0.0.1
```

Стандартный вывод

После перекодировки входных данных в CGI-модуле необходимо сформировать HTML-документ для передачи его на стандартное устройство вывода. При этом HTML-страница динамически генерируется на основании информации, хранимой в БД, которая может обновляться в реальном масштабе времени.

Интерфейс программирования серверных приложений ISAPI

Перспективным направлением, в котором развиваются технологии создания моделей расширения сервера, является интерфейс (протокол) программирования серверных приложений ISAPI. Этот интерфейс разработан для преодоления ограничений и недостатков интерфейса CGI. В отличие от CGI, интерфейс ISAPI предусматривает реализацию модулей расширения сервера в виде *динамической библиотеки*. При этом для обработки каждого запроса обозревателя к модулю расширения сервера ISAPI не создается отдельный процесс, а вызываются функции модуля ISAPI, которые один раз загружаются в память при первом обращении.

Есть несколько вариантов реализации протоколов серверных расширений. Наибольшее распространение получили два из них — NSAPI компании Netscape и ISAPI фирмы Microsoft. Из-за ограниченности объема книги мы рассмотрим только возможности протокола ISAPI.

Приложения, построенные с использованием протокола ISAPI, основаны на аналогичных с CGI-модулями принципах взаимодействия с сервером. Отличия указанных протоколов заключаются в осуществлении обмена данными между модулем и сервером.

Модули расширения ISAPI, как и модули CGI, принимают от сервера данные, которые передает модулю обозреватель, обрабатывают эти данные и динамически формируют HTML-документ. Однако вместо чтения содержимого переменных окружения и стандартного потока ввода модуль расширения ISAPI получает данные при помощи специально предназначенных для этого функций. Аналогично вместо записи выходных данных в стандартный поток вывода модуль расширения ISAPI также вызывает специальные функции.

Модули ISAPI в зависимости от назначения подразделяются на две группы: обычные модули расширения ISAPI и фильтры ISAPI. Первая группа предназначена, как и CGI-модули расширения сервера, для динамического формирования HTML-документов. Фильтры ISAPI обеспечивают выполнение нестандартных задач управления всеми данными, проходящими через сервер (например, кодирование данных, сбор статистической информации об использовании ресурсов сервера и т. д.).

Рассмотрим основные принципы создания и работы модулей расширений ISAPI. Модуль ISAPI реализуется в виде библиотеки DLL. Эта библиотека загружается в одно адресное пространство с Web-сервером при первом обращении к ней из обозревателя. Причем вызов ISAPI-модуля осуществляется подобно вызовам CGI-модулей расширения — из форм или ссылок. ISAPI-модуль исполняется в рамках единого адресного пространства программных ресурсов сервера. Такая организация ISAPI-модуля позволяет значительно повысить производительность сервера в целом по сравнению с сервером, использующим CGI-модули.

Преимущество ISAPI-интерфейса становится особенно заметным при одновременном активном использовании модуля расширения несколькими пользователями.

Отметим, что CGI-модули являются более надежным средством с точки зрения устойчивости Web-сервера к сбоям. Так, при возникновении ошибки в ISAPI-модуле может произойти аварийное завершение работы всего Web-сервера, тогда как в случае возникновения ошибки в CGI-модуле будет завершен только отдельный процесс. По упомянутой причине особенности отладки ISAPI-модуля состоят в том, что для перезапуска ISAPI-модуля требуется его переименовать или перезагрузить Web-сервер.

Рассмотрим структуру модулей ISAPI. Библиотека DLL, представляющая собой ISAPI-модуль, экспортирует как минимум две функции: `GetExtensionVersion` и `HttpExtensionProc`. Кроме этих функций, ISAPI-модуль может использовать функцию `TerminateExtension`, которой передается управление перед выгрузкой ISAPI-модуля из памяти. Эта функция освобождает ресурсы, выделенные ISAPI-модулю.

Функция `GetExtensionVersion(var Ver: THSE_VERSION_INFO): BOOL` предназначена для того, чтобы ISAPI-модуль мог сообщить серверу версию спецификации ISAPI-интерфейса и строку описания ISAPI-модуля. Параметр-переменная `Ver` указывает управляющую структуру, тип которой описан так:

```
type
  THSE_VERSION_INFO = packed record
    // Версия спецификации интерфейса ISAPI
    dwExtensionVersion: DWORD;
    // Строка описания модуля ISAPI
    lpszExtensionDesc: array [0..HSE_MAX_EXT_DLL_NAME_LEN-1] of Char;
  end;
```

Функция `HttpExtensionProc(var ECB: TEXTENSION_CONTROL_BLOCK): DWORD` обеспечивает основную функциональность ISAPI-модуля и используется для обмена данными между ISAPI-модулем и сервером. Параметр-переменная `ECB` указывает управляющую структуру, тип которой описан так:

```
type
  TEXTENSION_CONTROL_BLOCK = packed record
    // Размер блока в байтах
    cbSize: DWORD;
    // Версия спецификации интерфейса ISAPI
    dwVersion: DWORD;
    // Идентификатор соединения
    ConnID: HCONN;
    // Код состояния HTTP
    dwHttpStatusCode: DWORD;
```

```
    // Данные протоколирования
    lpszLogData: array [0..HSE_LOG_BUFFER_LEN — 1] of Char;
    // Переменная окружения REQUEST_METHOD
    lpszMethod: PChar;
    // Переменная окружения QUERY_STRING
    lpszQueryString: PChar;
    // Переменная окружения PATH_INFO
    lpszPathInfo: PChar;
    // Переменная окружения PATH_TRANSLATED
    lpszPathTranslated: PChar;
    // Полный размер данных, полученных от обозревателя
    cbTotalBytes: DWORD;
    // Размер доступного блока данных
    cbAvailable: DWORD;
    // Указатель на доступный блок данных
    lpbData: Pointer;
    // Тип полученных данных
    lpszContentType: PChar;
    // Функция для получения значений переменных окружения
    GetServerVariable: TGetServerVariableProc;
    // Функция для посылки данных обозревателю
    WriteClient: TWriteClientProc;
    // Функция для получения данных от посетителя
    ReadClient: TReadClientProc;
    // Служебная функция
    ServerSupportFunction: TServerSupportFunctionProc;
  end;
```

Поясним особенности использования основных полей этой структуры данных.

Поле `cbTotalBytes` предназначено для записи суммарного количества байтов данных, принимаемых от обозревателя. Блок данных размером не более 48 Кбайт считывается сервером автоматически. Эти данные становятся доступными при вызове функции `HttpExtensionProc`. Но полностью данные дочитываются в цикле при помощи функции `ReadClient`.

Поле `cbAvailable` служит для сохранения длины блока данных, полученных автоматически от обозревателя (не более 48 Кбайт).

Поле `LpbData` содержит указатель на область памяти, в которой размещен полученный сервером от обозревателя блок данных размером `cbAvailable` байтов.

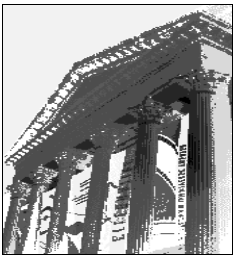
В управляющий блок помимо данных включены следующие указатели на методы.

Поле `GetServerVariable` содержит указатель на функцию, средствами которой ISAPI-модуль определяет значения переменных среды окружения.

В поле `WriteClient` находится указатель на метод, используемый для отправки данных обозревателю, в отличие от интерфейса CGI, в случае применения которого модуль расширения помещает результат в стандартный поток вывода.

Поле `ReadClient` хранит указатель на функцию, предназначенную для считывания данных в буфер предварительного чтения, имеющий адрес `LpbData` и размер, не превышающий 48 Кбайт.

В поле `ServerSupportFunction` содержится указатель на функцию, служащую для пересылки стандартного заголовка протокола HTTP и осуществления других действий.



ГЛАВА 33

Публикация баз данных средствами Delphi

Система Delphi предоставляет программисту мощные средства для разработки сложных Web-приложений. Специальные компоненты Delphi позволяют помещать информацию из БД непосредственно на HTML-страницы. С помощью Delphi можно создавать Web-приложения на основе интерфейсов CGI, WinCGI, ISAPI и NSAPI. В этой главе рассматривается использование интерфейсов CGI и ISAPI как наиболее часто встречающихся.

Компоненты, используемые при разработке Web-приложений

При разработке Web-приложений используются компоненты Палитры компонентов, расположенные на странице **Internet** (рис. 33.1). Рассмотрим назначение этих компонентов.

Рис. 33.1. Страница **Internet** Палитры компонентов

- `WebDispatcher` (Web-диспетчер) предназначен для создания Web-модуля данных и позволяет модулю расширения Web-сервера обрабатывать HTTP-запросы.
- `PageProducer` (генератор страниц) предназначен для преобразования HTML-шаблона в HTML-документ. При этом в HTML-шаблоне производится замена специальных тегов на данные, которые генерируются динамически при выполнении программы с помощью обработки события `OnHTMLTag`.
- `DataSetTableProducer` (генератор таблицы набора данных) предназначен для генерации HTML-документа, который содержит таблицу из данных, полученных в результате обработки запроса к БД на получение всех записей. Наследует свойства объекта `TDataSet`.

- `DataSetPageProducer` (генератор страницы набора данных) предназначен для преобразования HTML-шаблона, содержащего ссылки на поля БД, вместо которых динамически при выполнении приложения подставляются значения из текущей записи БД.
- `QueryTableProducer` (генератор таблицы по запросу) предназначен для генерации HTML-документа, который содержит таблицу из данных, полученных в результате обработки запроса к БД.
- `SQLQueryTableProducer` (генератор таблицы по SQL-запросу) предназначен для генерации HTML-документа, который содержит таблицу из данных, полученных в результате обработки SQL-запроса к БД.
- `TcpClient` (клиент TCP) добавляет объект типа `TTcpClient` к форме или модулю данных для преобразования приложения в TCP/IP-клиента; определяет требуемое соединение с сервером TCP/IP, управляет открытым соединением и завершает законченное соединение.
- `TcpServer` (сервер TCP) добавляет объект типа `TTcpServer` к форме или модулю данных для преобразования приложения в сервер TCP/IP; ожидает запросы на TCP/IP-соединения от других машин и устанавливает соединения при получении запросов.
- `UdpSocket` (сокет UDP) добавляет объект типа `TUdpSocket` к форме или модулю данных для преобразования приложения в UDP/IP-клиент или UDP/IP-сервер.
- `XMLDocument` (документ XML). Класс `TXMLDocument` может использоваться непосредственно для доступа к узлам (элементам) документа с помощью интерфейса `IXMLNode` или применяться совместно с классами и интерфейсами, генерируемыми мастером XML Data Binding.
- `WebBrowser` (Web-обозреватель) создает собственный обозреватель для отображения HTML-страниц. Для использования этого компонента требуется Microsoft Internet Explorer версии не ниже 4.0.

Замечание

UDP (User Datagram Protocol) представляет собой протокол передачи прикладных пакетов дейтаграммным методом. Подобно TCP, использует протокол IP для доставки данных, но, в отличие от TCP, обеспечивает обмен дейтаграммами без подтверждения.

В справке указано назначение еще двух компонентов: `ClientSocket` (сокет клиента, служит для создания приложений TCP/IP-клиента, может быть добавлен к форме или модулю данных) и `ServerSocket` (сокет сервера, служит для создания приложений TCP/IP-сервера, может быть добавлен к форме или модулю данных), которых на странице **Internet** Палитры в действительности нет.

При разработке Web-приложений наиболее часто используются компоненты, предназначенные для генерации HTML-документов (`PageProducer`, `QueryTableProducer`, `DataSetTableProducer`, `DataSetPageProducer`) и для создания модулей расширения Web-сервера (`WebDispatcher`). Более подробно особенности их применения будут рассмотрены далее.

Статическая публикация

Как отмечалось, статическая публикация базы данных в Интернете используется при редком обновлении информации в БД. При использовании такого типа публикации информация из БД помещается в HTML-документ, который хранится постоянно в файле на диске. При этом запрос на получение данных не приводит к большой загрузке сервера, т. к. пользователю возвращается уже готовый HTML-документ.

Для статической публикации можно использовать обычное приложение Windows, которое запускают периодически (после обновления информации в БД). Вообще говоря, для такой публикации достаточно создать простейшее приложение БД, которое обеспечивает:

- доступ к БД;
- генерацию HTML-документа на основе информации, получаемой из БД;
- сохранение созданного файла на диске.

Рассмотрим особенности использования средств Delphi для решения этих задач на следующем примере.

Пусть требуется выполнить статическую публикацию содержимого демонстрационной БД, входящей в состав стандартной поставки Delphi. Эта БД имеет псевдоним `DBDEMOS` и имя файла таблицы Animals.dbf (при полной установке Delphi псевдоним создается автоматически).

Разместим в форме 2 кнопки `Button` и установим с помощью Инспектора объектов их свойства `Name` в значения `Gen` и `Show` соответственно. Кроме кнопок, поместим в форму компонент `Table`. В листинге 33.1 приведен код модуля `DbHTML` генератора HTML-страниц.

Листинг 33.1. Пример генератора HTML-страниц

```
unit DbHTML;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Db, DBTables, StdCtrls, ComCtrls;
type
  TDbHTMLForm = class(TForm)
    Table1: TTable;
       Gen: TButton;
      Show: TButton;
    procedure GenClick(Sender: TObject);
    procedure ShowClick(Sender: TObject);
  end;
var
  DbHTMLForm: TDbHTMLForm;
implementation

{$R *.DFM}

uses
  Shellapi, FileCtrl;
```

```
// Обработчик события OnClick кнопки Gen
procedure TDbHTMLForm.GenClick(Sender: TObject);
var HTMLStr: TStringList;
          I: Integer;
begin
  // Инициализация объекта Table1
  Table1.Close;
  Table1.DatabaseName := 'DBDEMOS';
  Table1.TableName := 'Animals.dbf';
  Table1.Open;

  // Создание заголовка HTML-документа
  HTMLStr := TStringList.Create;
  HtmlStr.Clear;
  HtmlStr.Add ('<HTML>');
  HtmlStr.Add ('<HEAD>');
  HtmlStr.Add ('<TITLE>' + 'HTML-страница, отчет из БД' + '</TITLE>');
  HtmlStr.Add ('</HEAD>');
  HtmlStr.Add ('<BODY BGCOLOR="#FFFFEE">');
  HtmlStr.Add ('<H1><CENTER> Данные из файла ' + Table1.TableName +
               '</CENTER></H1>');

  // Вывод тега, открывающего таблицу
  HtmlStr.Add('<TABLE BORDER>');

  // Вывод тега, открывающего строку таблицы
  HtmlStr.Add('<TR>');
  // Создание заголовков таблицы
  for I := 0 to Table1.FieldCount - 1 do
    HtmlStr.Add('<TH>' + Table1.Fields[I].FieldName + '</TH>');
  // Вывод тега, закрывающего строку таблицы
  HtmlStr.Add('</TR>');
  // Вывод данных
  Table1.First;
  // Перебор всех записей
  while not Table1.EOF do begin
    HtmlStr.Add('<TR>');
    for I := 0 to Table1.FieldCount - 1 do
      // Вывод тега ячейки данных таблицы
      if Table1.Fields[I].DisplayText = ''
         then HtmlStr.Add('<TD>' + '____'+ '</TD>')
         else HtmlStr.Add('<TD>' + Table1.Fields[I].DisplayText + '</TD>');
    HtmlStr.Add('</TR>');
    Table1.Next;
  end;

  // Вывод тега, закрывающего таблицу
  HtmlStr.Add('</TABLE>');
  // Вывод тега, закрывающего тело документа
  HtmlStr.Add ('</BODY>');
  // Вывод тега, закрывающего документ
  HtmlStr.Add ('</HTML>');
```

```
  // Сохранение сформированного HTML-документа в файле Htmltest.htm
  HtmlStr.SaveToFile ('Htmltest.htm');
  HtmlStr.Free;
end;

// Обработчик события OnClick кнопки Show
procedure TDbHTMLForm.ShowClick(Sender: TObject);
begin
  // Запуск просмотра файла HTMLTEST.htm
  ShellExecute (Handle, 'open', pChar('HTMLTEST.htm'), '', '', sw_ShowNormal);
end;

end.
```

В приведенном примере HTML-страницы генерируются без использования специальных средств библиотеки VCL. Формирование и сохранение в файле HTML-страниц задается в явном виде. Интерфейсная часть приложения представляет собой окно с двумя кнопками: `Gen` и `Show`.

Рассмотрим подробнее, как происходит обработка события нажатия кнопки.

В процедуре `GenClick` используется объект `HTMLStr` типа `TStringList`, в котором хранятся строки формируемого HTML-документа. В начале процедуры выполняется инициализация объекта `Table1` с помощью следующего кода:

```
Table1.Close;
Table1.DatabaseName := 'DBDEMOS';
Table1.TableName := 'Animals.dbf';
Table1.Open;
```

Здесь для объекта `Table1` устанавливаются значения свойств, задающих источник БД. Свойство `DatabaseName` определяет псевдоним (`DBDEMOS`), а свойство `TableName` задает таблицу БД (в рассматриваемом случае используемая таблица находится в файле Animals.dbf). Методы `Close` и `Open` соответственно закрывают и устанавливают соединение с БД. При этом вызов метода `Close` перед инициализацией объекта `Table1` обеспечивает успешное выполнение кода при повторном обращении к процедуре `GenClick`.

Далее в процедуре осуществляются создание объекта `StringList` с помощью конструктора `Create`:

```
HTMLStr := TStringList.Create;
```

и его инициализация с помощью метода `Clear`:

```
HtmlStr.Clear;
```

Формирование HTML-документа выполняется с помощью метода `Add`:

```
HtmlStr.Add ('<HTML>');
```

Информация из БД выводится в формируемый HTML-документ в табличном виде. В частности, заголовок таблицы выводится с помощью следующего кода:

```
HtmlStr.Add('<TABLE BORDER>');
HtmlStr.Add('<TR>');
```

```
for I := 0 to Table1.FieldCount – 1 do
    HtmlStr.Add('<TH>' + Table1.Fields[I].FieldName + '</TH>');
HtmlStr.Add('</TR>');
```

Свойство `FieldCount` задает число полей в записи таблицы. Строка `Table1.Fields[I].FieldName` возвращает указатель на строку, содержащую имя поля. При этом объект `Fields` является массивом полей записи. Выражение `Fields[I]` позволяет обратиться к полю записи с номером `I` (нумерация с нуля).

Далее в обработчике стоит итерационный цикл, содержащий вложенный цикл с параметром:

```
Table1.First;
while not Table1.EOF do begin
  HtmlStr.Add('<TR>');
  for I := 0 to Table1.FieldCount – 1 do
      if Table1.Fields[I].DisplayText = ''
         then HtmlStr.Add('<TD>' + '____'+ '</TD>')
         else HtmlStr.Add('<TD>' + Table1.Fields[I].DisplayText + '</TD>');
  HtmlStr.Add('</TR>');
  Table1.Next;
end;
```

Перед началом цикла с помощью метода `First` указатель текущей записи устанавливается на первую запись таблицы. Во внешнем цикле указатель текущей записи последовательно перемещается по записям таблицы, а во внутреннем цикле осуществляется перебор всех полей текущей записи. Выход из внешнего цикла происходит при достижении указателем текущей записи конца таблицы (значение выражения `not Table1.EOF` становится ложным).

Метод `Table1.Fields[I].DisplayText` возвращает указатель на строку, содержащую данные из `I`-го поля текущей записи. Чтобы обозреватель не нарушал структуру таблицы, при отсутствии в поле данных в формируемый документ выводится строка `'<TD>' + '____'+'</TD>'` (если данные есть, то выводится `'<TD>'+Table1.Fields[I].DisplayText + '</TD>'`).

Последняя строка в теле цикла содержит вызов метода `Next`, который переводит указатель текущей записи на следующую запись в таблице.

Сохранение сформированного HTML-документа в файле Htmltest.htm на диске выполняется с помощью метода `SaveToFile` объекта `StringList`:

```
HtmlStr.SaveToFile ('Htmltest.htm');
```

В процедуре `ShowClick` вызывается API-функция Windows `ShellExecute`:

```
ShellExecute (Handle, 'open', pChar ('Htmltest.htm'), '', '',      sw_ShowNormal);
```

которая загружает файл Htmltest.htm в обозреватель.

Вид сформированного приложением HTML-документа в окне обозревателя показан на рис. 33.2. Как видно из рисунка, данные в HTML-документе представлены в виде таблицы, столбцы которой соответствуют полям, а строки — записям БД.

Рис. 33.2. Вид сгенерированного приложением HTML-документа в окне обозревателя

Компоненты генерации HTML-страниц

В следующих разделах рассматриваются специальные компоненты `PageProducer`, `DataSetPageProducer`, `DataSetTableProducer` и `QueryTableProducer`, предназначенные для эффективной работы с HTML-страницами в Delphi, а также пример генератора HTML-страниц.

Компонент *PageProducer*

Компонент `PageProducer` генерирует HTML-страницы на основе специального HTML-шаблона, связанного с этим компонентом. *HTML-шаблон* представляет собой последовательность HTML-команд и специальных тегов-транспарантов, имеющих следующий формат:

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

Содержимое шаблона может формироваться заранее (на этапе проектирования) или динамически — перед генерацией HTML-страницы.

Для доступа к шаблону используется свойство `HTMLDoc` типа `TStrings` или `HTMLFile` типа `TFileName`. Свойства `HTMLDoc` и `HTMLFile` являются взаимоисключающими — при выборе одного из них другое свойство обнуляется.

При вызове метода `Content` производится синтаксический анализ HTML-строк. При обработке подлежащих замене тегов возникает событие `OnHTMLTag` типа `THTMLTagEvent`, в обработчике которого производится замена всех тегов-транспарантов на необходимые текстовые значения. Кроме того, метод `Content` возвращает указатель на буфер, в котором находится сформированный документ. Например:

```
Memo1.Text := PageProducer1.Content;
```

Свойство Dispatcher типа TCustomWebDispatcher компонента PageProducer служит для связи с Web-сервером. Оно содержит ссылку на Web-модуль WebModule или WebDispatcher, в функции которых входит обслуживание текущего запроса. Особенности работы с этим свойством рассматриваются далее.

Компонент *DataSetPageProducer*

Компонент DataSetPageProducer аналогичен предыдущему компоненту, но дополнительно содержит свойство DataSet типа TdataSet, в котором указывается используемый набор данных. При применении этого компонента можно в тегах-транспарантах HTML-шаблона (связанного с компонентом DataSetPageProducer через свойства HTMLDoc или HTMLFile) в качестве имени тега (TagName) указывать имена полей в записи БД. Тем самым обеспечивается отображение значений полей текущей записи в определяемом свойством DataSet наборе данных. Например:

```
Имя:    <#name>   <BR>
Размер: <#size>   <BR>
Вес:    <#weight> <BR>
Место:  <#area>   <BR>
```

Здесь name, size, weight, area — имена полей в записи БД.

Следующий фрагмент кода формирует в редакторе Memo1 HTML-документ по данным первой записи из БД:

```
Table1.First;
Memo1.Clear;
Memo1.Text := DataSetPageProducer1.Content;
```

Инструкция Table1.First устанавливает указатель текущей записи на первую запись набора данных, инструкция Memo1.Clear очищает все строки редактора Memo1, после чего он заполняется содержимым сформированного HTML-документа с помощью метода Content объекта DataSetPageProducer1.

Компонент *DataSetTableProducer*

Компонент DataSetTableProducer позволяет выводить содержимое БД в табличном виде. Рассмотрим наиболее часто используемые свойства и методы этого компонента.

Свойство Header типа TStrings задает заголовок HTML-документа, а свойство Footer типа TStrings задает строки, заканчивающие HTML-документ.

Свойство DataSet определяет набор данных, на основании которого формируется HTML-документ.

Свойство RowAttributes типа THTMLTableRowAttributes позволяет отдельно форматировать строки таблицы, а свойство TableAttributes типа THTMLTableAttributes служит для форматирования всей таблицы.

Метод Content компонента DataSetTableProducer, как и одноименный метод перечисленных выше компонентов, предназначен для генерации документа. Однако в отличие от предыдущих компонентов, событие OnHTMLTag в компоненте DataSetTableProducer не

генерируется. Кроме того, при форматировании ячейки таблицы возникает событие `OnFormatCell` типа `THTMLFormatCellEvent`, обработчик которого позволяет индивидуально форматировать каждую ячейку. Тип события описан следующим образом:

```
type THTMLFormatCellEvent =
  procedure THTMLForm.DataSetTableProducer1FormatCell(Sender: TObject;
      CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
      var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs,
      CellData: String);
```

Здесь параметры `CellRow` и `CellColumn` содержат номера редактируемой строки и столбцы таблицы соответственно, при этом параметр `CellRow` имеет значение `0` при формировании заголовка таблицы, а нулевое значение `CellColumn` соответствует формированию первого столбца таблицы.

Параметры `BgColor`, `Align` и `VAlign` задают цвет фона ячейки и горизонтальное и вертикальное выравнивание. Поскольку они входят в процедуру с описателем `var`, их значения можно изменить программно. Например:

```
if (CellRow = 0) then BgColor:='gray'
   else if (CellRow > 5) BgColor:='silver';
```

Допускается присваивать параметру `Align` значения `haLeft`, `haRight`, `haCenter`, что обеспечивает выравнивание влево, вправо и по центру ячейки соответственно.

Параметр `VAlign` может принимать значения `haTop`, `haMiddle`, `haBottom`, `haBaseline` (выравнивание по верху, посередине, по низу и по базовой линии в ячейке соответственно).

Параметр `CellData` содержит выводимые в ячейке данные. Этот параметр можно программно задавать для каждой ячейки. Например:

```
if (CellColumn > 2) then
   CellData := '<FONT size=6>' + CellData + '</FONT>';
```

Свойство `Columns` компонента `DataSetTableProducer` связано с Редактором столбцов. При двойном щелчке мыши в Инспекторе объектов на свойстве `Columns` или на значке компонента `DataSetTableProducer` в рабочей форме открывается окно Редактора столбцов (рис. 33.3). Отметим, что Редактор столбцов не имеет ничего общего с Редактором, используемым для настройки столбцов компонента `DBGrid`.

Загрузить Редактор столбцов можно также, выбрав пункт **Response Editor** контекстного меню компонента `DataSetTableProducer`.

Замечание

Информация из БД становится доступной в Редакторе столбцов, если в связанном с компонентом `DataSetTableProducer` объекте `Table` свойство `Active` установлено в значение `True`.

В верхней части окна Редактора столбцов находится панель с кнопками для управления порядком вывода полей БД в результирующую HTML-таблицу. Кроме того, можно изменять количество и порядок следования полей БД с помощью команд контекстного меню Редактора столбцов.

Перечислим назначение некоторых кнопок панели инструментов:

- **Add All Fields** — добавляет все поля таблицы, при этом появляется запрос на удаление уже присутствующих полей;
- **Delete Selected** — удаляет выбранные поля;
- **Add New** — добавляет новые поля;
- **Move Selected Up** и **Move Selected Down** — перемещают выбранное поле вверх или вниз;
- **Restore Defaults** — устанавливает для выбранного поля таблицы значения свойств форматирования по умолчанию.

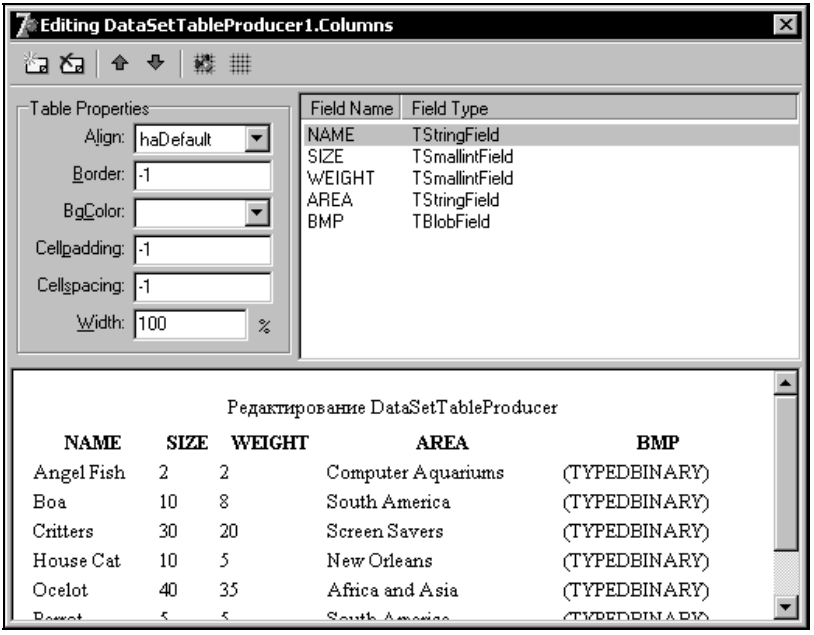


Рис. 33.3. Вид Редактора столбцов компонента `DataSetTableProducer`

При успешном открытии таблицы БД ее поля отображаются в нижней части Редактора столбцов.

Редактор столбцов позволяет определять группу выводимых столбцов и параметры таблицы с помощью средств визуального программирования. При этом создается HTML-код, в котором атрибутам табличных тегов присваиваются соответствующие значения параметров, установленные в Редакторе столбцов.

С помощью панели **Table Properties** можно задать следующие свойства, относящиеся ко всем полям таблицы:

- `Align` — горизонтальное выравнивание текста в ячейке таблицы, допустимые значения: `haDefault`, `haCenter`, `haLeft`, `haRight`;
- `Border` — толщина рамки (при значении `Border`, равном `-1`, рамка не выводится);

- BgColor — цвет фона ячеек таблицы, может принимать значения констант стандартных цветов, используемых в Delphi; по умолчанию цвет ячеек совпадает с цветом фона HTML-страницы;
- CellPadding — отступ в пикселах между текстом ячейки и рамкой таблицы;
- CellSpacing — отступ между ячейками таблицы в пикселах;
- Width — ширина таблицы в процентах от ширины HTML-страницы.

Редактирование отдельного столбца таблицы происходит с одновременным использованием Редактора столбцов и Инспектора объектов. При выделении поля в правом верхнем окне Редактора столбцов в Инспекторе объектов становятся доступными свойства этого поля. Задание свойств Align и BgColor для столбца аналогично их заданию для всей таблицы. Кроме того, становятся доступными дополнительные свойства. Вот наиболее часто используемые из них:

- FieldName — имя поля в таблице БД;
- Title — настройка свойств заголовка столбца, имеет аналогичные всей таблице свойства Align и BgColor. Свойство Caption задает заголовок столбца, свойство Custom добавляет дополнительные атрибуты к генерируемому элементу HTML, а свойство VAlign определяет вертикальное выравнивание заголовка и может принимать значения haDefault, haTop, haBottom, haMiddle, haBaseLine.

Компонент *QueryTableProducer*

Компонент QueryTableProducer отличается от компонента DataSetTableProducer наличием свойства Query типа TQuery, которое используется для выполнения SQL-запроса к базе данных.

Пример генератора HTML-страниц

В примере выполняется формирование статического HTML-документа, аналогичного документу из *разд. "Статическая публикация" данной главы*. Однако здесь генерация документа происходит с использованием стандартных компонентов Delphi.

В форме находятся четыре кнопки с именами GenPageProducer, GenDataSetPageProducer, GenTableProducer, GenQueryTableProducer, предназначенные для генерации HTML-документа с помощью размещенных в форме компонентов PageProducer, DataSetPageProducer, DataSetTableProducer и QueryTableProducer соответственно. Для отображения сформированной HTML-страницы в обозревателе предназначена кнопка с именем ShowHtml. В форме также размещены многострочный редактор Memo1 и наборы данных Table1 и Query1.

В качестве источника данных для компонентов DataSetPageProducer и GenTableProducer используется набор данных Table1, связанный с этими компонентами через их свойство DataSet.

В качестве значения свойства HTMLDoc компонента PageProducer установлен следующий текст:

```
<TITLE>Демонстрация возможностей PageProducer </TITLE>
</HEAD><BODY>
```

```
<H1> Демонстрация PageProducer </H1>
Эта страница демонстрирует возможности PageProducer <P>
Имя базы данных <B> <#dbname> </B>
Дата: <B> <#date> </B>. <P>
<HR>
</BODY></HTML>
```

При формировании HTML-документа (методом `Content`) теги-транспаранты `<#dbname>` и `<#date>` в этом шаблоне будут заменены на конкретные значения.

Значением свойства `HTMLDoc` компонента `DataSetPageProducer` является следующий текст:

```
<HTML> <HEAD>
<TITLE> Данные из файла <#dbname> </TITLE>
</HEAD> <BODY>
<H1> <CENTER> Данные из файла <#dbname> </CENTER> </H1>
Данные первой записи из БД <BR>
Имя:    <#name>   <BR>
Размер: <#size>   <BR>
Вес:    <#weight> <BR>
Место:  <#area>   <BR>
Дата:   <#date>   <BR>
</BODY> </HTML>
```

Здесь теги-транспаранты `<#dbname>`, `<#name>`, `<#size>`, `<#weight>`, `<#area>` и `<#date>` при формировании HTML-документа также будут заменены на конкретные значения полей БД.

В качестве источника данных для компонента `QueryTableProducer` используется набор данных `Query1`, связанный с этим компонентом через свойство `DataSet` последнего. Свойству `SQL` объекта `Query1`, используемому для хранения SQL-запроса к БД, необходимо задать значение, которое обеспечит получение требуемых данных. В рассматриваемом примере свойство `SQL` объекта `Query1` содержит следующий запрос:

```
SELECT * FROM Authors
WHERE YearBorn > 1960
```

В листинге 33.2 приведен код модуля `DBHForm` главной формы `HTMLForm` приложения.

Листинг 33.2. Пример генератора HTML-страниц

```
unit DBHForm;
interface
uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms, DBCtrls,
  StdCtrls, DBTables, ExtCtrls, Mask, Db, Dialogs, HTTPApp, DSProd, DBWeb;
type
  THTMLForm = class(TForm)
                ShowHtml: TButton;
         GenPageProducer: TButton;
 GenDataSetPageProducer: TButton;
        GenTableProducer: TButton;
```

```
  GenQueryTableProducer: TButton;
                  Memo1: TMemo;
          PageProducer1: TPageProducer;
   DataSetPageProducer1: TDataSetPageProducer;
  DataSetTableProducer1: TDataSetTableProducer;
    QueryTableProducer1: TQueryTableProducer;
                 Table1: TTable;
                 Query1: TQuery;

    procedure FormCreate(Sender: TObject);
    procedure GenPageProducerClick(Sender: TObject);
    procedure GenDataPageProducerClick(Sender: TObject);
    procedure GenTableProducerClick(Sender: TObject);
    procedure GenQueryTableProducerClick(Sender: TObject);
    procedure ShowHtmlClick(Sender: TObject);
    procedure DataSetPageProducer1HTMLTag(Sender: TObject; Tag: TTag;
      const TagString: String; TagParams: TStrings;
      var ReplaceText: String);
    procedure PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
      const TagString: String; TagParams: TStrings;
      var ReplaceText: String);
    procedure DataSetTableProducer1FormatCell(Sender: TObject;
      CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
      var Align: THTMLAlign; var VAlign: THTMLVAlign; CustomAttrs,
      var CellData: String);
  end;
var
  HTMLForm: THTMLForm;
implementation
{$R *.DFM}
uses
  ShellAPI;
procedure THTMLForm.FormCreate(Sender: TObject);
begin
  // Инициализация наборов данных
  Table1.Close;
  Table1.DatabaseName := 'DBDEMOS';
  Table1.TableName := 'Animals.dbf';
  Table1.Open;
  Query1.DatabaseName := 'Authors';
  Query1.Open;
end;

procedure THTMLForm.ShowHtmlClick(Sender: TObject);
begin
  // Сохранение сгенерированного HTML-документа
  Memo1.Lines.SaveToFile ('htmltest.htm');
  // Загрузка HTML-документа в обозреватель
  ShellExecute(Handle, 'open', PChar('htmltest.htm'),
               '', '', sw_ShowNormal);
end;
```

```
// Генерация HTML-страницы компонента PageProducer
procedure THTMLForm.GenPageProducerClick(Sender: TObject);
begin
  Memo1.Clear;
  Memo1.Text := PageProducer1.Content;
end;

// Генерация HTML-страницы компонента DataSetPageProducer
procedure THTMLForm.GenDataSetPageProducerClick(Sender: TObject);
begin
  Table1.First;
  Memo1.Clear;
  Memo1.Text := DataSetPageProducer1.Content;
end;

// Генерация HTML-страницы компонента TablePageProducer
procedure THTMLForm.GenTableProducerClick(Sender: TObject);
begin
  Memo1.Clear;
  Memo1.Text := DataSetTableProducer1.Content;
end;

// Генерация HTML-страницы компонента QueryTableProducer
procedure THTMLForm.GenQueryTableProducerClick(Sender: TObject);
begin
  Memo1.Clear;
  Memo1.Text := QueryTableProducer1.Content;
end;

// Обработка события OnHTMLTag компонента PageProducer1,
// используемая для замены тегов-транспарантов
procedure THTMLForm.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
                      const TagString: String; TagParams: TStrings;
                      var ReplaceText: String);
begin
  // Замена тегов <#dbname> и <#date>
  if TagString = 'dbname'
    then ReplaceText := Table1.TableName
    else if TagString = 'date' then ReplaceText := DateToStr(Date);
end;

// Обработка события OnHTMLTag компонента DataSetPageProducer1,
// используемая для замены тегов-транспарантов
procedure THTMLForm.DataSetPageProducer1HTMLTag(Sender: TObject;
                      Tag: TTag;
                      const TagString: String; TagParams: TStrings;
                      var ReplaceText: String);
begin
  // Замена тегов <#dbname> и <#date>
  if TagString = 'dbname'
    then ReplaceText := Table1.TableName
    else if TagString = 'date' then ReplaceText := DateToStr(Date);
```

```
  // Замена тегов <#name>, <#size>, <#weight>, <#area> на значения
  // соответствующих полей текущей записи БД производится автоматически
end;

// Обработка события OnFormatCell компонента DataSetTableProducer1
procedure THTMLForm.DataSetTableProducer1FormatCell(Sender: TObject;
            CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
            var Align: THTMLAlign; var VAlign: THTMLVAlign;
            var CustomAttrs, CellData: String);
begin
  if CellColumn > 2
    then CellData := '<FONT size=6>' + CellData + '</FONT>';
  if Length(CellData) > 2
    then CellData := '<B>' + CellData + '</B>';
  if CellRow > 5 BgColor := 'silver';
end;
end.
```

Инициализация наборов данных `Table1` и `Query1` осуществляется в процедуре `FormCreate`. Псевдоним `Authors` используется для связи набора данных `Query1` с БД, входящей в состав учебника по ASP-страницам, устанавливаемого вместе с операционной системой Windows 2000 Server. Этот псевдоним необходимо создать самостоятельно с помощью Администратора источников данных ODBC, установив связь псевдонима `Authors` с указанной БД. Напомним, что названный Администратор входит в состав утилит Панели управления операционной системы.

Для формирования HTML-документа с использованием компонента `PageProducer` предназначена кнопка `GenPageProducer`. Вид полученного HTML-документа в окне обозревателя показан на рис. 33.4.

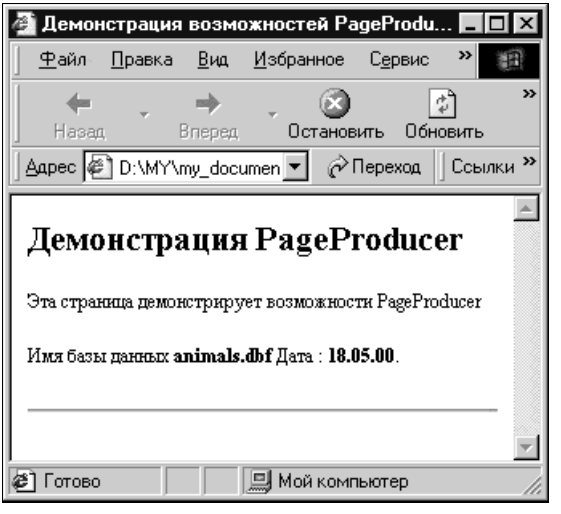


Рис. 33.4. HTML-страница, сгенерированная с помощью компонента `PageProducer`

HTML-страница содержит статический текст, состоящий из имени таблицы БД и строки с обозначением текущего времени. Страница формируется при вызове метода `Content`, при этом генерируется событие `OnHTMLTag` и выполняется его обработчик (процедура `PageProducer1HTMLTag`). В коде обработчика происходит замена тегов-транспарантов на значения имени таблицы БД и текущего времени, осуществляемая на основании анализа шаблона HTML-документа:

```
if TagString = 'dbname'
    then ReplaceText := Table1.TableName
    else if TagString = 'date' then ReplaceText := DateToStr(Date);
```

Здесь параметр `TagString` обработчика задает имя тега, при обработке которого произошло событие `OnHTMLTag`, а параметр `ReplaceText` содержит текст, на который заменяется тег.

Формирование HTML-документа с использованием компонента `DataSetPageProducer`, обращающегося к данным компонента `Table1`, происходит при нажатии кнопки `GenDataSetPageProducer`. Вид полученного HTML-документа в окне обозревателя показан на рис. 33.5.

Рис. 33.5. HTML-страница, полученная с использованием компонента `DataSetPageProducer`

На этой странице выводится информация из первой записи БД. Для установки указателя на первую запись используется метод `First` набора данных `Table1`.

В процессе генерации HTML-документа возникает событие `OnHTMLTag`. Обработчик этого события для компонента `DataSetTableProducer` в основном совпадает с обработчиком аналогичного события рассмотренного выше компонента `PageProducer`, однако замена специальных тегов, имеющих имена, совпадающие с именами соответствующих полей таблицы БД, на значения этих полей производится автоматически.

Для формирования HTML-документа с помощью компонента `DataSetTableProducer` предназначена кнопка `GenTableProducer`. Вид полученного HTML-документа в окне обозревателя показан на рис. 33.6.

Показанный HTML-документ содержит данные таблицы БД, находящиеся в файле Animals.dbf. Формирование документа выполняется вызовом метода `Content` компонента `DataSetTableProducer`. При генерации HTML-документа используются параметры, установленные в Редакторе столбцов. При форматировании каждой ячейки таблицы возникает событие `OnFormatCell`, в обработчике которого редактируются параметры отображения.

В рассматриваемом примере форматирование заключается в следующем:

- устанавливается шестой размер шрифта для ячеек четвертого и следующих столбцов таблицы:

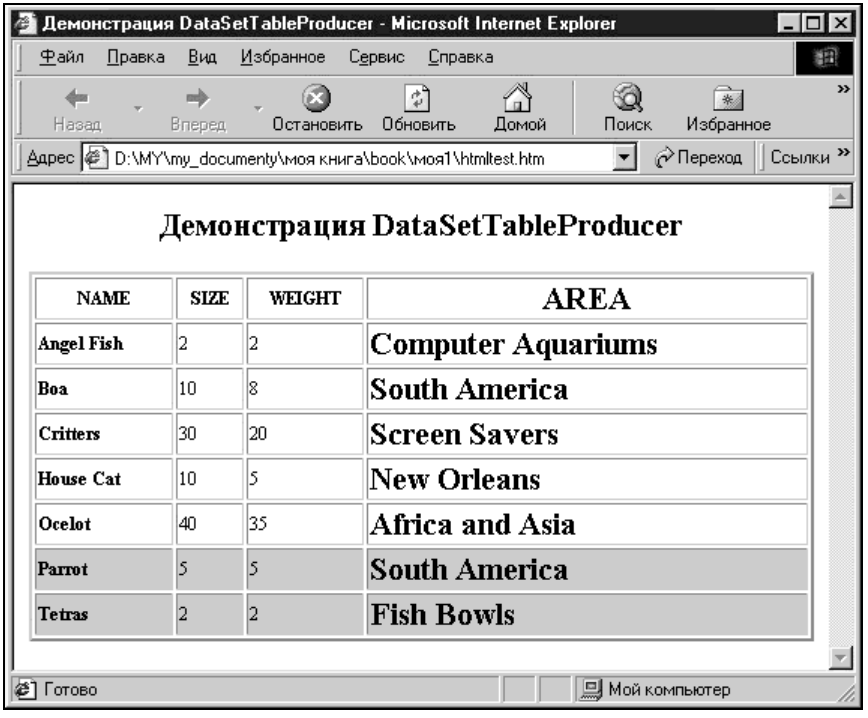
```
CellData := '<FONT size=6>' + CellData + '</FONT>';
```

- устанавливается полужирный шрифт для ячеек, размер содержимого которых превышает 2:

```
CellData := '<B>' + CellData + '</B>';
```

- для строк, начиная с 6-й, задается серебристый цвет фона:

```
BgColor := 'silver';
```



Демонстрация DataSetTableProducer

NAME	SIZE	WEIGHT	AREA
Angel Fish	2	2	Computer Aquariums
Boa	10	8	South America
Critters	30	20	Screen Savers
House Cat	10	5	New Orleans
Ocelot	40	35	Africa and Asia
Parrot	5	5	South America
Tetras	2	2	Fish Bowls

Рис. 33.6. HTML-страница,
полученная с использованием компонента `DataSetTableProducer`

Для формирования HTML-документа с помощью компонента `QueryTableProducer` используется кнопка `GenQueryTableProducer`. Компонент `QueryTableProducer` отличается от

DataSetTableProducer тем, что для получения данных применяется набор данных Query, а не Table. Вид сгенерированного HTML-документа в окне обозревателя показан на рис. 33.7.

В приведенном HTML-документе содержатся записи из таблицы Authors, отбор которых обеспечивает компонент Query1. Его SQL-запрос выбирает записи, для которых значение года превышает 1960.

Перейдем теперь к описанию динамической публикации, которая, как уже говорилось, используется для БД, обновляемых в реальном масштабе времени. Примером таких БД являются интернет-магазины, информационные системы в крупных интрасетях и т. д.

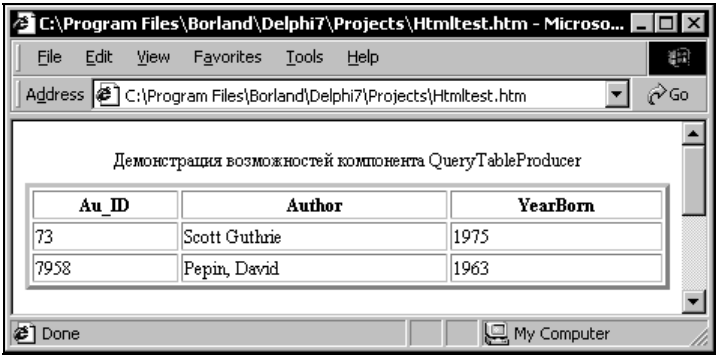


Рис. 33.7. HTML-страница, полученная с использованием компонента QueryTableProducer

Динамическая публикация

Для Web-приложений, использующих БД, могут применяться технологии активных серверных страниц (ASP), модули расширения на основе Java или интерфейсов CGI, ISAPI/NSAPI, WinCGI. При этом применение интерфейсов является предпочтительным по сравнению с другими технологиями при решении нестандартных задач, например, при обеспечении взаимодействия Web-сервера с сервером, осуществляющим прием платежей для обслуживания интернет-магазина. Кроме того, модули расширения сервера, созданные на основе интерфейсов, позволяют реализовать более быструю обработку запросов пользователей Web-приложения к БД.

В этом разделе рассматриваются средства Delphi, предназначенные для создания модулей расширения на основе интерфейсов CGI и ISAPI.

Создание модуля CGI

Познакомимся с особенностями разработки модуля CGI средствами Delphi на примере обычного консольного приложения.

Создадим модуль CGI с именем cgifirst.exe, который будет осуществлять генерацию HTML-страниц на основе информации из базы данных Animals.dbf, входящей в демонстрационный пакет программ Delphi.

Для запуска этого модуля нужно выполнить следующие действия:

1. Установить Web-сервер (Personal WS или MS IIS).
2. Задать в параметрах проекта путь для выходного каталога (OutPut), где размещаются сценарии (для Personal WS — это каталог Webshare\Scripts, а для MS IIS — Inetpub\Scripts).
3. Создать в среде Delphi новое консольное приложение.

Загрузка модуля cgifirst.exe производится с помощью HTML-страницы, содержащей ссылку:

```
<P> <A HREF="http://igin/scripts/cgifirst.exe">
  Пример CGI-модуля </A> </P>
```

Исходный текст программы (консольного приложения) `CgiFirst` приводится в листинге 33.3.

Листинг 33.3. Пример консольного приложения для создания модуля CGI

```
program CgiFirst;
{$APPTYPE CONSOLE}
uses
  Windows, SysUtils, DBTables, DB;
var
  // Массив для хранения URL CGI-модуля
  ScriptName: array [0..100] of Char;
  // Массив для хранения параметров строки запроса
  PathName: array [0..30] of Char;
  // Массив для хранения типа метода
  MethodName: array [0..30] of Char;
  // Флаг, используемый для возвращения на главную страницу
  k: Integer;
  // Внутренняя переменная
      ii: Integer;
  Table1: TTable;

// Процедура вывода заголовка страницы
procedure Header; forward;

// Процедура вывода окончания страницы
procedure Footer; forward;

// Процедура вывода списка ссылок
procedure ShowList; forward;

// Процедура вывода таблицы данных
procedure ShowTable; forward;

// Процедура вывода результатов запроса
procedure Showquery(it: Integer); forward;
```

```
// Вывод заголовка страницы
procedure Header;
begin
  writeln('Content type: text/html');
  writeln;
  writeln('<HTML> <HEAD>');
  writeln('<TITLE> Cgi-приложение </TITLE>');
  writeln('</HEAD> <BODY>');
  writeln('<H1> Публикация БД CGI-приложением </H1>');
  writeln('<HR>');
end;

// Вывод окончания страницы
procedure Footer;
begin
  writeln('<HR>');
  writeln('<CENTER> <B> БД из файла ' +
          Table1.TableName + '</B> </H2> </CENTER>');
  writeln('</BODY>');
  writeln('</HTML>');
end;

// Вывод списка ссылок
procedure ShowList;
var i: Integer;
begin
  Table1.First;
  // Вывод заголовка списка
  writeln('<CAPTION> <B> <FONT size=6> Список животных' +
          '</FONT> </B> </CAPTION>');
  writeln('<P> <TABLE BORDER>');
  writeln('<TR> <TH>' + 'Данные' + '</TH>');
  // Вывод списка без последнего столбца
  for i := 0 to Table1.FieldCount — 2 do begin
    writeln('<TH>');
    writeln('<A HREF="' + ScriptName + '/' +
       Table1.Fields[i].FieldName + '"> <B>' +
       Table1.Fields[i].FieldName + '</B> </A>');
    writeln('</TH>');
  end;
  writeln('</TR> </TABLE>');
  writeln('<P> <P> <TABLE BORDER>');
  writeln('<TR> <TH>');
  writeln('<A HREF="' + ScriptName +
          '"> <B> Вернуться назад на основную страницу </B> </A>');
  writeln('</TR> </TH>');
  writeln('</TABLE>');
end;

// Вывод таблицы данных
procedure ShowTable;
var i: Integer;
```

```
begin
  Table1.First;
  writeln('<CAPTION> <B> <FONT SIZE=6>' +
          'Таблица с данными о животных </FONT> </B> </CAPTION>');
  writeln('<P>< TABLE BORDER>');
  writeln('<TR>');
  // Вывод заголовков таблицы
  for i := 0 to Table1.FieldCount — 2 do
    writeln('<TH>' + Table1.Fields [i].FieldName + '</TH>');
  writeln('</TR>');
  // Вывод строк таблицы
  while not Table1.EOF do begin
    writeln('<TR>');
    for i := 0 to Table1.FieldCount — 2 do
      writeln('<TD>' + Table1.Fields [i].AsString + '</TD>');
    writeln('</TR>');
    Table1.Next;
  end;
  writeln('</TABLE>');
  writeln('<P> <P> <TABLE BORDER>');
  writeln('<TR> <TH>');
  // Вывод ссылки на запуск cgifirst.exe без параметров
  writeln('<A HREF="'+ScriptName +
          '"> <B> Вернуться назад на основную страницу </B> </A>');
  writeln('</TR> </TH>');
  writeln('</TABLE>');
end;

// Вывод результатов запроса
procedure Showquery(it: Integer);
var i: Integer;
begin
  Table1.First;
  writeln('<CAPTION> <B> <FONT SIZE=6>' +
          'Данные о животных </FONT> </B> </CAPTION>');
  writeln('<P> <TABLE BORDER>');
  writeln('<TR>');
  // Вывод заголовка списка
  writeln('<TH>' + Table1.Fields[it].FieldName + '</TH>');
  // Вывод данных требуемого столбца
  while not Table1.EOF do begin
    writeln('<TD>' + Table1.Fields [it].AsString + '</TD>');
    Table1.Next;
  end;
  writeln('</TR>');
  writeln('</TABLE>');
  writeln('<P> <P> <TABLE BORDER>');
  writeln('<TR> <TH>');
  // Вывод специальной ссылки, которая при ее выборе (событие
  // onClick) возвращает предыдущую страницу
  // ("window.history.back()" — текст на JavaScript)
```

```
  writeln('<A HREF="" onClick="window.history.back()">' + '
          '<B> Вернуться назад </B> </A>');
  writeln('</TR> </TH>');
  writeln('<TR> <TH>');
  writeln('<A HREF="'+ScriptName +
          '"> <B> Вернуться назад на основную страницу </B> </A>');
  writeln('</TR> </TH>');
  writeln('</TABLE>');
end;

// Главная программа
begin
  // Вывод заголовка страницы
  Header;
  // Считывание переменных окружения
  // URL
  GetEnvironmentVariable('SCRIPT_NAME', ScriptName, sizeof(ScriptName));
  // Параметры из командной строки
  GetEnvironmentVariable('PATH_INFO', PathName, sizeof(PathName));
  // Создание таблицы
  Table1 := TTable.Create(nil);
  try
    Table1.DatabaseName := 'DBDEMOS';
    Table1.TableName := 'Animals.dbf';
    Table1.Open;
    k := 0;
    // Анализ параметров запроса
    if StrComp(PathName, '/List') = 0 then begin
      k := 1;
      ShowList;
    end
    else if StrComp(PathName, '/Table') = 0
           then begin ShowTable; k := 1; end
           else begin
             for ii := 0 to Table1.FieldCount — 2 do begin
               if StrComp (PathName,Pchar('/' +
                  Table1.Fields [ii].FieldName)) = 0) then
                     begin Showquery(ii);k := 1 end
             end;
           end;
    if k = 0 then begin
       writeln('<A HREF="' + ScriptName + '/List"> <B>' +
               'Загрузить список животных </B> </A> <P>');
       writeln('<A HREF="' + ScriptName +'/TABLE"> <B>' +
               'Загрузить данные о животных </B> </A> <P>');
    end;
    // Вывод окончания страницы
    Footer;
  finally
    Table1.Close;
    Table1.Free;
  end;
end.
```

В зависимости от параметров запроса приведенная программа обеспечивает вывод четырех HTML-страниц. При обработке на сервере URL-запроса вида

```
http://igin/scripts/cgifirst.exe
```

выводится главная страница (первая), содержащая следующие две гиперссылки на подчиненные страницы:

```
Загрузить список животных
Загрузить данные о животных
```

Первая из этих гиперссылок связана с URL-адресом **http://igin/scripts/cgifirst.exe/List** и адресует ко второй HTML-странице, вид которой показан на рис. 33.8.

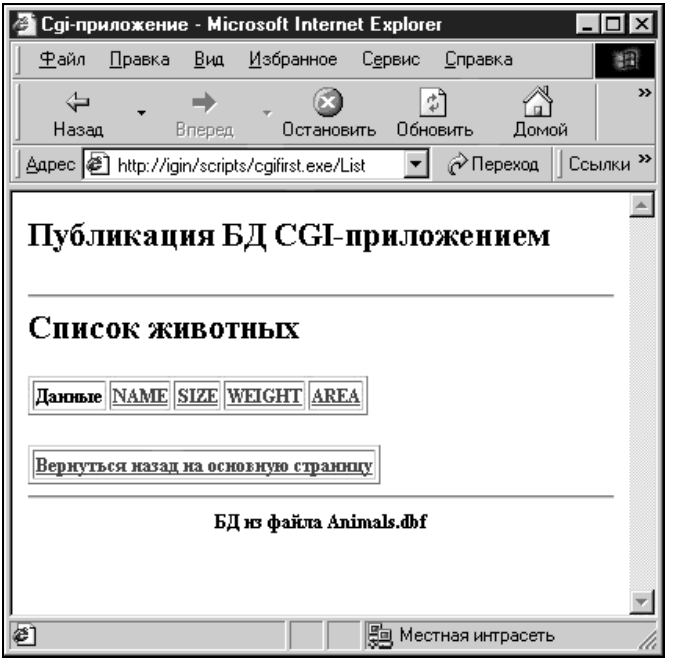


Рис. 33.8. Вид первой подчиненной HTML-страницы

Вторая гиперссылка содержит URL **http://igin/scripts/cgifirst.exe/Table** и адресует к третьей HTML-странице (рис. 33.9).

Четвертая HTML-страница вызывается при выборе гиперссылок, содержащихся в ячейках верхней таблицы (см. рис. 33.8), причем строка запроса в URL параметризуется в зависимости от выбранного в этой таблице поля. Например, для первого поля (NAME) URL принимает значение **http://igin/scripts/cgifirst.exe/NAME** и адресует к HTML-странице, показанной на рис. 33.10.

Таким образом, приведенное CGI-приложение обеспечивает формирование четырех страниц при обработке на стороне сервера следующих URL-запросов:

- **http://igin/scripts/cgifirst.exe** — главная страница, выполняется основная часть программы;

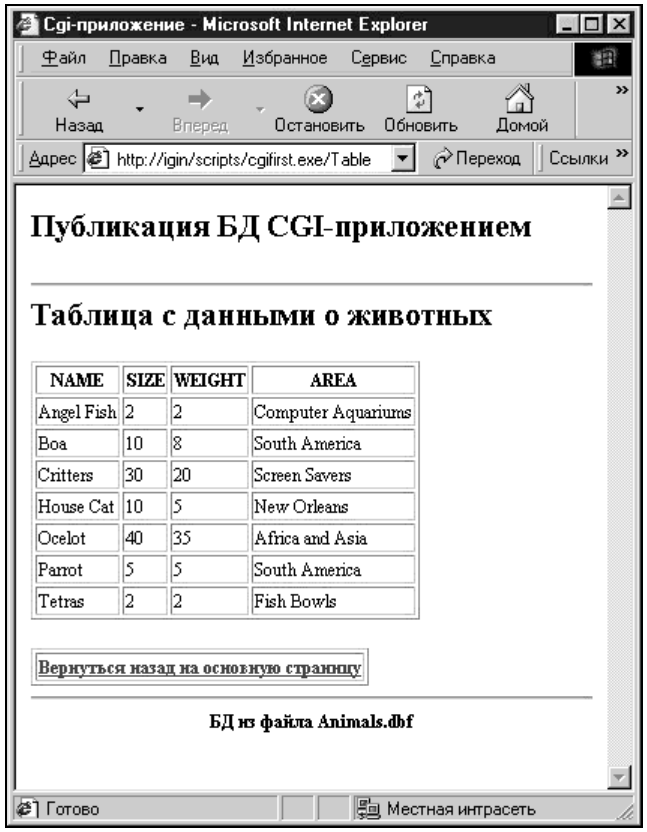


Рис. 33.9. Вид второй подчиненной HTML-страницы

- **http://igin/scripts/cgifirst.exe/List** — первая подчиненная таблица (см. рис. 33.8), выполняется процедура `ShowList`;
- **http://igin/scripts/cgifirst.exe/Table** — вторая подчиненная таблица (см. рис. 33.9), выполняется процедура `ShowTable`;
- **http://igin/scripts/cgifirst.exe/NAME** — подчиненная страница нижнего уровня (рис. 33.10); кроме параметра `NAME`, в запросе могут использоваться названия и других полей таблицы — `SIZE`, `WEIGHT`, `AREA`, в этом случае загружающая эту страницу ссылка может изменяться.

Поясним назначение наиболее важных частей программы. В начале основной части программы вызывается процедура `Header`, которая выводит заголовок для каждой HTML-страницы.

Для доступа к переменным окружения используется функция Windows API, имеющая следующий прототип:

```
DWORD GetEnvironmentVariable(
    LPCTSTR lpName,    // Адрес имени переменной окружения
    LPTSTR  lpBuffer,  // Адрес буфера для значения переменной
    DWORD   nSize      // Размер буфера в символах
  );
```

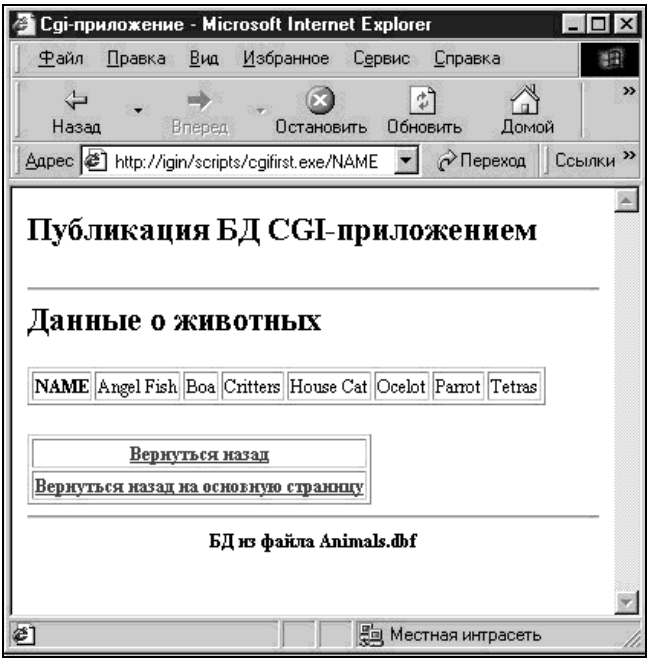


Рис. 33.10. Вид подчиненной HTML-страницы нижнего уровня

В программе эта функция вызывается дважды — для считывания значений системных переменных `SCRIPT_NAME` и `PATH_INFO`. Переменная `SCRIPT_NAME` содержит URL-адрес (без подстроки параметров) выполняемого в данный момент модуля расширения, а переменная `PATH_INFO` содержит подстроку передаваемых параметров, которая входит в состав URL-адреса после символа `/`. Например, для URL-адреса **http://igin/scripts/cgifirst.exe/NAME** переменная `SCRIPT_NAME` содержит подстроку `http://igin/scripts/cgifirst.exe`, а переменная `PATH_INFO` — подстроку `/NAME`.

Для доступа к данным используется переменная `Table1` типа `TTable`. Управление объектом `Table1` осуществляет следующий код:

```
Table1 := TTable.Create (nil);
try
    Table1.DatabaseName := 'DBDEMOS';
    Table1.TableName := 'Animals.dbf';
    Table1.Open;
...
finally
    Table1.Close;
    Table1.Free;
end;
```

Здесь метод `Create` класса `TTable` применяется для создания экземпляра объекта `Table1`. Для обработки исключений, связанных с открытием БД, используется конструкция `try...finally`.

Для определения страницы, которую должен формировать CGI-модуль, служит функция StrComp, которая сравнивает значение параметра URL-запроса, хранящегося в переменной PathName, с соответствующей строкой. При совпадении содержимого переменной PathName со строкой /List для генерации HTML-документа вызывается процедура ShowList, а при совпадении содержимого переменной PathName со строкой /Table — процедура ShowTable.

В цикле проверяется совпадение значения переменной PathName со строкой, содержащей названия полей таблицы БД:

```
for ii := 0 to Table1.FieldCount — 2 do begin
  if StrComp(PathName, Pchar('/' + Table1.Fields [ii].FieldName)) = 0 then
        begin Showquery(ii); k := 1; end
```

При совпадении вызывается процедура Showquery, в которую передается индекс соответствующего поля. Процедура PChar возвращает указатель на строку

```
'/' + Table1.Fields[ii].FieldName
```

Если ни одна из строк не совпадает с содержимым переменной PathName, то загружается главная страница CGI-приложения.

Для возврата на главную страницу (верхнего уровня) используется ссылка вида:

```
<A HREF="http://igin/scripts/cgifirst.exe">
<B> Вернуться назад на основную страницу </B> </A>
```

Возврат на предыдущие страницы осуществляется по ссылке:

```
<A HREF="" onClick="window.history.back()">
<B> Вернуться назад </B> </A>
```

Выбор этой гиперссылки вызывает событие onClick, при обработке которого выполняется функция JavaScript.back() объекта window.history.

Создание ISAPI-модуля расширения сервера

Web-приложение, выступающее в роли модуля расширения сервера и поддерживающее протокол программирования ISAPI, является динамически загружаемой библиотекой (DLL). Delphi обладает удобными средствами визуального проектирования таких приложений.

Рассмотрим схему работы Web-приложения, которое создано с помощью средств Delphi и является модулем расширения сервера (рис. 33.11).

При получении Web-приложением HTTP-запроса создается объект WebRequest, который обрабатывается в Web-модуле WebModule. Этот объект является связующим звеном между сервером и модулем расширения.

Для обработки запроса в модуле WebModule имеется набор объектов WebActionItems. Модуль WebModule определяет, какой из этих объектов отвечает за обработку поступившего запроса, при этом вызывается соответствующий обработчик события OnActions, в котором для ответа формируется объект WebResponse.

Модуль WebModule сравнивает значения свойств MethodType и Pathinfo объектов WebActionItem с соответствующими значениями параметров в HTTP-запросе. При сов-

падении значений этих свойств вызывается обработчик события `OnAction` соответствующего объекта `WebActionItem`. При этом свойству `Enable` объекта `WebActionItem` в процессе проектирования должно быть присвоено значение `True`. Если ни у одного из объектов `WebActionItem` свойства `MethodType` и `Pathinfo` не совпали с соответствующими значениями параметров в HTTP-запросе, то модуль расширения прекращает свою работу, и сервер прерывает соединение с клиентом.

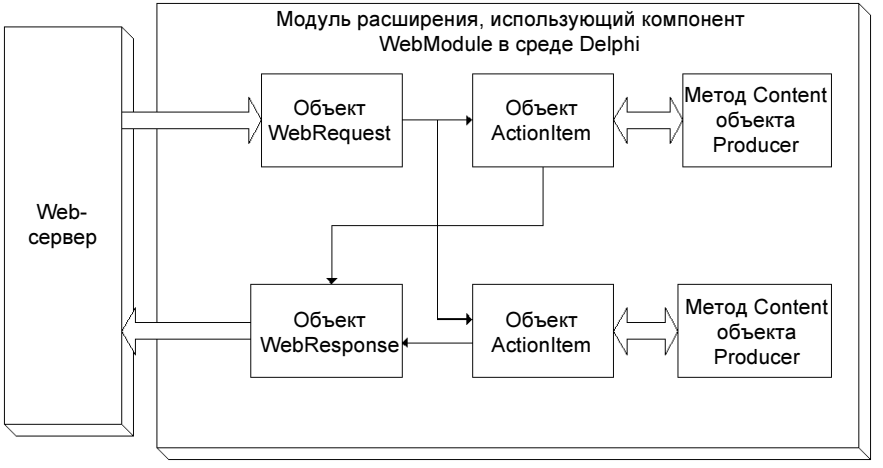


Рис. 33.11. Схема взаимодействия объектов модуля расширения сервера

В случае использования компонентов-генераторов HTML-страниц для обработки запроса может вызываться их метод `Content`, который помещает формируемый HTML-документ в объект `WebResponse`.

Перечислим свойства объектов `WebRequest` и `WebResponse`, которые могут потребоваться при создании обычного Web-приложения.

Для получения передаваемых параметров из обозревателя в модуле расширения сервера `WebModule` используются следующие свойства объекта `WebRequest`:

- `URL` типа `String` (URL-запрос);
- `ScriptName` типа `String` (URL-путь к приложению, выполняемому на сервере);
- `QueryFields` типа `String` (параметры запроса, переданные в строке URL);
- `ContentFields` типа `String` (параметры запроса при использовании методов `Post` или `Put`).

Возвращение серверу сформированного HTML-документа осуществляется с помощью перечисленных ниже свойств объекта `WebResponse`.

- `Content` типа `String` (буфер для информации, возвращаемой обозревателю в качестве ответа на запрос);
- `ContentStream` типа `Stream` (указатель на поток, используемый для возвращения обозревателю большого объема информации);
- `ContentType` типа `String` (указатель типа возвращаемых обозревателю данных).

Замечание

ISAPI-модули расширения предназначены для многопользовательского доступа. При обработке нескольких запросов к одному и тому же модулю расширения управление передается единственному экземпляру ISAPI-модуля. Для обеспечения коллективного доступа к БД требуется объект `Session`. Если поместить в Web-модуль компонент класса `TSession` и установить его свойства `Active` и `AutoSessionName` в значение `True`, то для каждого нового запроса к БД будет устанавливаться новое подключение.

Рассмотрим создание простого Web-приложения, использующего интерфейс ISAPI.

Обратимся для этого к Хранилищу объектов и выберем на его странице **New** объект **Web Server Application**. После выбора этого объекта появляется окно (рис. 33.12) задания типа используемого серверного приложения (протокола), в нашем случае это `ISAPI/NSAPI`.

Рис. 33.12. Окно выбора типа серверного приложения

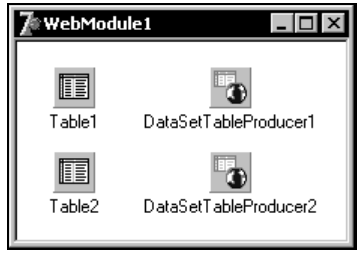


Рис. 33.13. Окно Web-модуля

В результате открывается окно Web-модуля **WebModule1** (рис. 33.13), в котором можно размещать требуемые компоненты. Разместим в модуле два набора данных `Table` и свяжем эти компоненты с таблицами демонстрационной БД. Для генерации HTML-документов добавим к модулю два компонента `DataSetTableProducer`.

Иерархию (дерево) составляющих модуль `WebModule` объектов можно отображать в окне **Object TreeView** (рис. 33.14). Здесь же можно удалить объекты, добавить и удалить элементы (поля) объектов, а также изменить порядок следования элементов.

Действия с объектами `WebActionItem` выполняются в окне редактирования действий модуля **Editing WebModule1.Actions** (рис. 33.15), открываемом двойным щелчком на свободном месте окна модуля командой **Action Editor** контекстного меню модуля или через его свойство `Actions` в Инспекторе объектов.

Добавим в этом окне требуемые действия ISAPI-модуля, осуществляемые в зависимости от параметров командной строки. Добавление действия выполняется с помощью крайней левой кнопки панели инструментов. Обработчик каждого параметра задается как обработчик события `OnActions` в каждом компоненте `WebActionItems`. Имя обработ-

чика события `OnActions` для объекта `WebActionItem` можно определить с помощью Инспектора объектов (рис. 33.16).

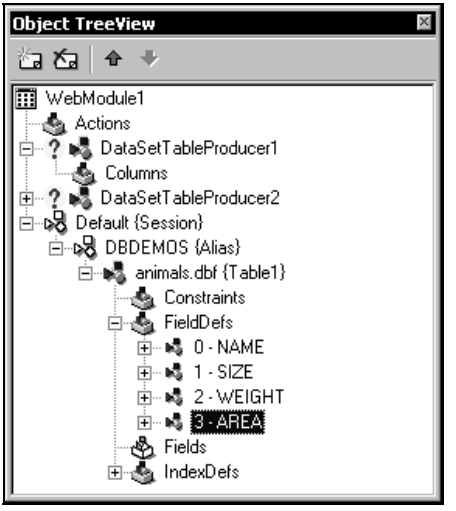


Рис. 33.14. Иерархия составляющих модуль `WebModule` компонентов

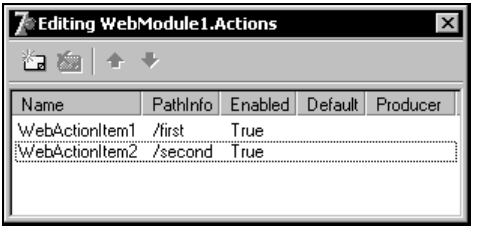


Рис. 33.15. Окно редактирования действий Web-модуля

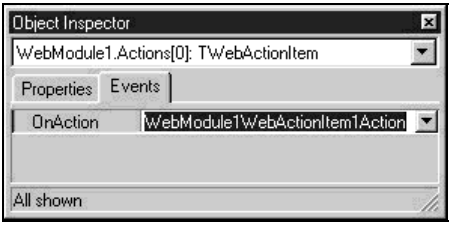


Рис. 33.16. Событие объекта `WebActionItem`

Для параметризации вызова модуля указываются значения свойств `PathInfo` для каждого объекта `WebActionItem` (также через Инспектор объектов).

Замечание

При отладке ISAPI-модуля возникает проблема, связанная с внесением изменений в соответствующую библиотеку DLL. Чтобы перегрузить созданную заново библиотеку, требуется перезапускать Web-сервер. Поэтому рекомендуется либо переименовывать ISAPI-модуль при повторном обращении к обновленной библиотеке, либо сначала создать для отладки CGI-приложение. После завершения отладки приложения нужно перекомпилировать его в ISAPI-модуль.

Модуль Dbmodule.dll выводит на HTML-страницы отчет из двух БД в виде таблиц. Для загрузки модуля используется следующая HTML-страница:

```
<HTML><HEAD>
</HEAD>
<BODY>
```

```
  <A HREF="http://igin/scripts/DBmodule.dll/First">
  Вывести БД из файла biolife.db </A> </P>
  <P> <A HREF="http://igin/scripts/DBmodule.dll/Second">
  Вывести БД из файла animals.dbf </A> </P>
</BODY>
</HTML>
```

Исходный текст модуля `Dbmodule` представлен в листинге 33.4.

Листинг 33.4. Пример вывода на HTML-страницу отчета из базы данных

```
unit Dbmodule;
interface
uses
  Windows, Messages, SysUtils, Classes, HTTPApp, DBWeb, Db, DBTables;
type
  TWebModule1 = class(TWebModule)
                  Table1: TTable;
    DataSetTableProducer1: TDataSetTableProducer;
                  Table2: TTable;
    DataSetTableProducer2: TDataSetTableProducer;

    procedure WebModule1WebActionItem1Action(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure WebModule1WebActionItem2Action(Sender: TObject;
              Request: TWebRequest; Response: TWebResponse;
              var Handled: Boolean);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  WebModule1: TWebModule1;
implementation
{$R *.DFM}

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
begin
  Response.Content := '<HTML><HEAD>' + '<TITLE>ISAPI-приложение</TITLE>' +
       '</HEAD> <BODY> <H1> БД из файла ' + Table1.TableName + ' </H1>';
  Response.Content := Response.Content + DataSetTableProducer1.Content
      + '</BODY> </HTML>';
end;

procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
                     Request: TWebRequest; Response: TWebResponse;
                     var Handled: Boolean);
```

```
begin
  Response.Content := '<HTML> <HEAD> <TITLE> ISAPI-приложение </TITLE>'
     +'</HEAD> <BODY> <H1> БД из файла ' + Table2.TableName+' </H1>';
  Response.Content := Response.Content + DataSetTableProducer2.Content;
  Response.Content := Response.Content + '</BODY> </HTML>';
end;
end.
```

При выборе первой ссылки будет загружаться HTML-страница, вид которой приведен на рис. 33.17.

При выборе второй ссылки выводится HTML-страница, содержащая отчет из БД biolife.dbf, по формату аналогичная первой странице.

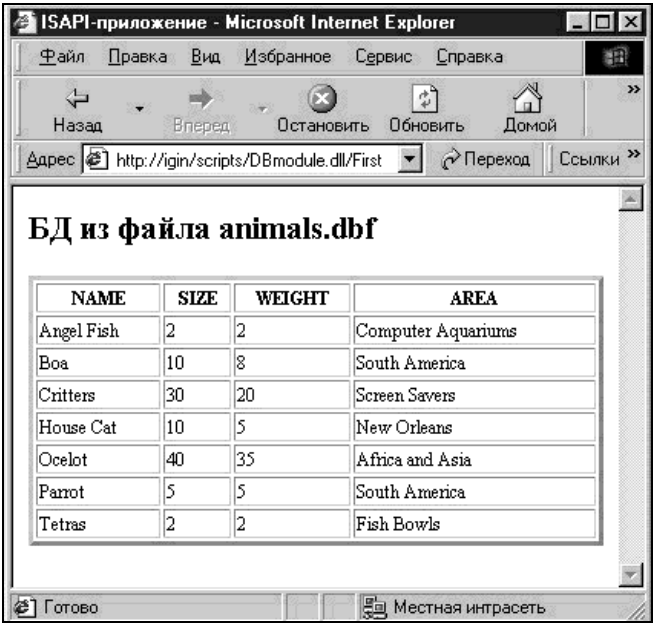


Рис. 33.17. Вывод модулем Dbmodule.dll данных из файла animals.dbf

Обработка пользовательского ввода в модуле ISAPI

При создании информационных систем на основе Web-приложений, публикующих БД в Интернете, для обеспечения взаимодействия с пользователем необходимо реализовать обработку вводимых им данных в формах на HTML-страницах. Получение передаваемых из обозревателя параметров наиболее часто выполняется с помощью методов `GET` и `POST`. Особенности работы сервера при использовании каждого из этих методов были описаны в *главе 32*.

В среде Delphi для получения передаваемых из обозревателя параметров используется объект типа `TWebRequest`. Рассмотрим наиболее важные свойства этого объекта.

Для получения параметров, переданных методом GET, предназначено свойство QueryFields типа TStrings. В этом свойстве переданные параметры записываются в следующем формате:

```
Параметр1=Значение_Параметра1
Параметр2=Значение_Параметра2
...
ПараметрN=Значение_ПараметраN
```

Доступ к данным, переданным методом GET, осуществляется с помощью следующей инструкции:

```
Response.Content := Request.QueryFields.Values['N1'];
```

где N1 — имя переданного параметра.

Для получения параметров, переданных методом POST, используется свойство ContentFields типа TStrings. Доступ к данным при этом происходит так:

```
Response.Content := Request.ContentFields.Values['N1'];
```

Рассмотрим особенности использования свойств объекта TWebRequest на примере. Создадим обычное ISAPI-приложение, содержащее два объекта WebActionItem, параметры PathInfo которых задаются равными \1 и \2 соответственно. Поместим на поле Web-диспетчера компоненты Query и QueryTableProducer и установим свойство Query объекта QueryTableProducer в значение Query1. С помощью Редактора столбцов установим параметр Border компонента QueryTableProducer в значение 5, что обеспечит отображение рамки вокруг таблицы.

Для ввода пользовательских данных может использоваться следующая HTML-страница (листинг 33.5).

Листинг 33.5. Пример HTML-страницы для ввода данных

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<BR>
<A HREF="http://localhost/Scripts/input.dll/1?
   N1=Ганс Андерсен&N2=1990">
Передача параметров методом GET
</A>
<FORM NAME=Form1 METHOD="Post"
ACTION="http://localhost/Scripts/Input.dll/2>
<BR>
Эта форма позволяет вводить новую строку в БД "Authors"
<BR>
Поля для ввода имени автора: <INPUT TYPE=Text NAME="N1"> <BR>
Поля для ввода года издания: <INPUT TYPE=Text NAME="N2"> <BR>
<BR>
<INPUT TYPE=Submit VALUE="Выполнить запрос">
</FORM>
</BODY>
</HTML>
```

Эта страница позволяет ввести два параметра: имя автора (`N1`) и год издания книги (`N2`).

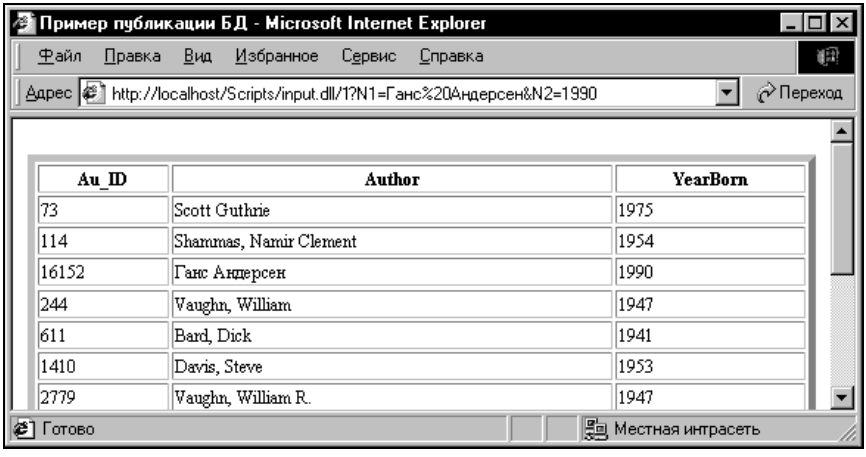
Для добавления записей используем БД, имеющую псевдоним `Authors`. Напомним, что в названной БД присутствуют поля `author` и `YearBorn`.

Следующий код, находящийся в процедуре `WebModule1WebActionItem2Action`, позволяет сформировать посылаемый с помощью объекта `Query1` запрос к БД на добавление записи:

```
Query1.Close;
Query1.DatabaseName := 'Authors';
Query1.SQL.Clear;
Query1.SQL.Add ('INSERT INTO authors(author, YearBorn) values (''' +
   Request.ContentFields.Values['N1'] + ''', ' +
   Request.ContentFields.Values['N2'] + ')';
Query1.ExecSQL;
Query1.SQL.Clear;
Query1.SQL.Add('SELECT * FROM authors');
Query1.Open;
Response.Content := '<HTML> <HEAD>' +
   '<TITLE> Пример публикации БД </TITLE>' +
   '</HEAD> <BODY>' + QueryTableProducer1.Content;
```

Процедура `WebModule1WebActionItem2Action` вызывается, если передача параметров осуществляется с применением метода `POST`, а если для передачи параметров используется метод `GET`, то в этом случае вызывается процедура `WebModule1WebActionItem1Action` с аналогичным кодом, за исключением того, что в нем вместо свойства `ContentFields` применяется свойство `QueryFields`.

Для запуска ISAPI-модуля с передачей параметров методом `POST` нужно загрузить приведенную ранее HTML-страницу, заполнить поля ввода и нажать кнопку **Выполнить запрос**. Запуск ISAPI-модуля с методом `GET` происходит при выборе ссылки **Передача параметров методом GET**.



Au_ID	Author	YearBorn
73	Scott Guthrie	1975
114	Shammas, Namir Clement	1954
16152	Ганс Андерсен	1990
244	Vaughn, William	1947
611	Bard, Dick	1941
1410	Davis, Steve	1953
2779	Vaughn, William R.	1947

Рис. 33.18. Окно обозревателя с результатом работы ISAPI-модуля

Вид окна обозревателя с результатом работы ISAPI-модуля с передачей параметров методом GET показан на рис. 33.18.

В результате обработки запроса обозревателя рассматриваемым ISAPI-модулем будет сформирован HTML-документ, содержащий в виде таблицы все строки обновленной БД.

Публикация графики

Рассмотрим применение стандартных средств Delphi для динамической публикации графической информации из БД на следующем примере.

Разрабатываемый модуль реализуем в виде ISAPI-библиотеки, использующей таблицу из файла animals.dbf. Для запуска данного приложения нужно создать ISAPI-модуль, как описано ранее, для которого потребуются 2 объекта WebActionItem. Свойства PathInfo этих объектов-действий устанавливаются в значения text и image, кроме того, подготавливаются обработчики для события OnAction. В модуле необходимо также разместить объекты DataSetTableProducer и Table.

В примере предварительно созданы три HTML-файла, текст которых приводится далее. Управляющий файл proton.html содержит следующий код:

```
<HTML>
<HEAD>
<TITLE> Демонстрация работы с графикой </TITLE>
</HEAD>
  <FRAMESET COLS="70%,30%">
    <FRAME SRC="main.html" NAME=main>
    <FRAME SRC="image.html" NAME=image>
  </FRAMESET>
</HTML>
```

Совместное отображение графики и текста в HTML-документе удобно реализовать с помощью кадров. Тег <FRAMESET COLS="70%,30%"> задает набор кадров. Параметр COLS используется для разбивки рабочей области обозревателя на два столбца шириной 70 и 30% соответственно.

Тег <FRAME SRC="main.html" name=main> определяет параметры кадра: параметр SRC задает отображаемый в кадре файл, а параметр name служит для доступа к созданным в этом кадре объектам.

Файл первого кадра (с именем main.html) содержит следующий код:

```
<HTML>
<BODY>
<SCRIPT>
  location="http://igin/scripts/isag.dll/text"
</SCRIPT>
</BODY>
</HTML>
```

В теге <SCRIPT> для передачи управления модулю расширения isag.dll используется свойство location объекта windows.

Файл второго кадра (с именем image.html) содержит код:

```
<HTML>
<HEAD>
<TITLE>Демонстрация работы с графикой</TITLE>
</HEAD>
<BODY>
<SCRIPT>
  location="http://igin/scripts/isag.dll/image?0"
</SCRIPT>
</BODY>
</HTML>
```

Исходный текст программы `isag` (табличное отображение отчета из БД) содержит следующий код (листинг 33.6).

Листинг 33.6. Пример табличного отображения отчета из базы данных

```
unit isag;
interface
uses
  Windows, Messages, SysUtils, Classes, HTTPApp, DBWeb, Db, DBTables;
type
  TWebModule1 = class(TWebModule)
                Table1: TTable;
  DataSetTableProducer1: TDataSetTableProducer;
  procedure DataSetTableProducer1FormatCell(Sender: TObject; CellRow,
                    CellColumn: Integer; var BgColor: THTMLBgColor;
                    var Align: THTMLAlign; var VAlign: THTMLVAlign;
                    var CustomAttrs, CellData: String);
  procedure WebModuleAfterDispatch(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
    procedure WebModuleBeforeDispatch(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
  procedure WebModule1WebActionItem1Action(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
  procedure WebModule1WebActionItem2Action(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
  private
    ScriptName: string;
end;
var
  WebModule1: TWebModule1;
implementation
{$R *.DFM}
uses
  Graphics, Jpeg, ExtCtrls,dbctrls;
```

```
// Обработчик события OnFormatCell
// Параметр Target указывает на соседний кадр
procedure TWebModule1.DataSetTableProducer1FormatCell(
        Sender: TObject; CellRow, CellColumn: Integer;
        var BgColor: THTMLBgColor; var Align: THTMLAlign;
        var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellColumn = 4) and (CellRow > 0) then
    CellData := '<A HREF="' + ScriptName + '/image?' +
      IntToSTR(CellRow — 1) + ' "Target="image"> Show Image </A>';
end;

// Обработчик события AfterDispatch
procedure TWebModule1.WebModuleAfterDispatch(Sender: TObject;
                  Request: TWebRequest; Response: TWebResponse;
                  var Handled: Boolean);
begin
  Table1.close;
end;

// Обработчик события BeforeDispatch
procedure TWebModule1.WebModuleBeforeDispatch(Sender: TObject;
                 Request: TWebRequest; Response: TWebResponse;
                 var Handled: Boolean);
begin
  Table1.Open;
  ScriptName := Request.ScriptName;
end;

// Обработчик события OnAction объекта WebActionItem1
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
                  Request: TWebRequest; Response: TWebResponse;
                  var Handled: Boolean);
begin
  // Формирование ответа обозревателю с помощью
  // компонента DataSetTableProducer1
  Response.Content:= DataSetTableProducer1.Content;
end;

// Обработчик события OnAction объекта WebActionItem2
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
                   Request: TWebRequest; Response: TWebResponse;
                   var Handled: Boolean);
var
   DataSource1: TDataSource;
      DBImage1: TDBImage;
         Jpeg1: TJpegImage;
        Stream: TMemoryStream;
           num: Integer;
begin
  DataSource1 := TDataSource.Create(nil);
  DBImage1 := TDBImage.Create(nil);
```

```
  try
    Table1.Open;
    DataSource1.DataSet := Table1;
    DBImage1.DataSource := DataSource1;
    DBImage1.DataField := 'BMP';
    // Извлечение параметров запроса
    num := StrToInt(Request.Query);
    if num = 0 then Table1.First
               else Table1.MoveBy(num);
    Jpeg1 := TJpegImage.Create;
    try
      Jpeg1.CompressionQuality := 50;
      Jpeg1.Assign(DBImage1.Picture.Bitmap);
      Stream := TMemoryStream.Create;
      try
        Jpeg1.SaveToStream (Stream);
        Stream.Position := 0;
        Response.ContentStream := Stream;
        Response.ContentType := 'image/jpeg';
        Response.SendResponse;
      finally
        Stream.Free;
      end;
    finally
      Jpeg1.Free;
    end;
  finally
    Table1.Close;
  end;
end;
end.
```

Результаты обработки запросов модулем isag.dll приведены на рис. 33.19.

В первом кадре (рис. 33.19) в виде таблицы отображаются текстовые данные. Для формирования таблицы в левом кадре используется следующий URL-запрос:

```
http://igin/scripts/isag.dll/text
```

Для обработки этого запроса вызывается процедура `WebModule1WebActionItem1Action`, в которой генерируется HTML-документ, содержащий таблицу. Формирование HTML-документа выполняет метод `Content` компонента `DataSetTableProducer1`:

```
Response.Content:=DataSetTableProducer1.Content;
```

При форматировании каждой ячейки этой таблицы в методе `Content` объекта `DataSetTableProducer1` возникает событие `OnFormatCell`, в обработчике которого содержится условная инструкция

```
if (CellColumn = 4) and (CellRow > 0) then ...
```

Тем самым допускается редактирование только пятого столбца всех строк, начиная со второй, и для них формируются ссылки вида

```
<A HREF="http://igin/scripts/isag.dll/image?0" TARGET="image"> Show Image </A>
```

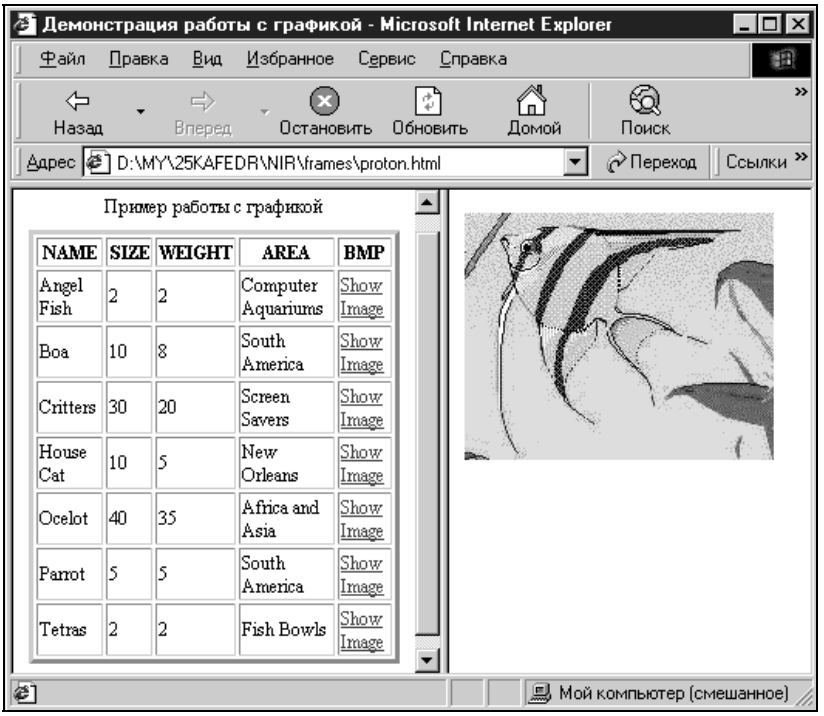


Рис. 33.19. Результаты работы модуля isag.dll

Цифра после знака `?` изменяется в зависимости от номера строки. Атрибут `TARGET="image"` задает целевой кадр для вывода графики после обработки запроса.

Собственно публикация графической информации осуществляется в процедуре `WebModule1WebActionItem2Action`.

Для извлечения параметров URL-запроса, передаваемых в самом запросе, используется свойство `Query` параметра `Request`. Для преобразования строкового значения переданного параметра применяется процедура `StrToInt`:

```
num := StrToInt(Request.Query);
```

С помощью метода `MoveBy` объекта `Table` указатель текущей записи устанавливается на запись номер `num`.

Графическая информация извлекается из БД с помощью объекта `DBImage`:

```
DataSource1 := TDataSource.Create(nil);
DBImage1 := TDBImage.Create(nil);
Table1.Open;
DataSource1.DataSet := Table1;
DBImage1.DataSource := DataSource1;
DBImage1.DataField := 'BMP';
```

Информация в поле `BMP` базы данных хранится в формате BMP. Чтобы обозреватель мог отобразить графическую информацию, ее необходимо преобразовать в формат JPG. Для этого используется объект класса `TJPEG`:

```
Jpeg1 := TJpegImage.Create;
Jpeg1.CompressionQuality := 50;
```

Преобразование файла из формата BMP в формат JPG осуществляется с помощью метода `Assign` объекта `JpegImage`:

```
Jpeg1.Assign(DBImage1.Picture.Bitmap);
```

Для отправки изображения обозревателю используется промежуточный поток:

```
Stream := TMemoryStream.Create;
Jpeg1.SaveToStream(Stream);
```

Указатель текущей позиции устанавливается на начало потока с помощью свойства `Position` объекта `Stream`:

```
Stream.Position := 0;
```

Далее указателю на возвращаемый поток Web-сервера присваивается значение указателя на промежуточный поток:

```
Response.ContentStream := Stream;
```

Тип возвращаемых данных устанавливается следующим образом:

```
Response.ContentType := 'image/jpeg';
```

С помощью метода `SendResponse` находящиеся в потоке данные отсылаются обозревателю:

```
Response.SendResponse;
```

Замечание

Событие `BeforeDispatch` возникает прежде, чем Web-диспетчер приступит к обработке поступившего HTTP-запроса. Событие `AfterDispatch` возникает после того, как HTTP-ответ успешно сформирован, но еще не отослан. Эти события удобно использовать для выполнения соответственно инициализации и удаления различных ресурсов, необходимых для обработки запроса.

Использование технологии ADO

В завершение главы рассмотрим средства доступа к источникам данных с использованием технологии (интерфейса) ADO, являющейся надстройкой над интерфейсом OLE DB. Напомним, что основа интерфейса ADO — набор объектов, на вершине иерархии которого находятся объекты `Connection`, `Command` и `Recordset`, обеспечивающие соединение с электронными источниками информации, поддерживающими этот интерфейс.

Для работы с интерфейсом ADO предназначены компоненты, находящиеся на странице **ADO** Палитры компонентов (см. рис. 14.8). Назначение указанных компонентов приводится в *главе 14*. Кроме того, работа с базами данных по технологии ADO освещалась в *главе 22*.

Рассмотрим особенности использования компонентов ADO на примере модуля ISAPI, публикующего БД в Интернете.

Для запуска примера требуется создать обычное ISAPI-приложение, как это было сделано в предыдущих разделах. После создания основного модуля, реализующего функции Web-диспетчера, добавим к нему один объект `WebActionItem`.

В данном приложении для получения доступа к источнику данных достаточно использовать два компонента: `ADOQuery` и `ADOConnection`. Поместим их на поле объекта `WebDispatcher`, добавим еще компонент `DataSetTableProducer`.

Установим свойство `DataSet` объекта `DataSetTableProducer1` в значение `ADOQuery1` и зададим рамку вокруг таблицы, установив с помощью Редактора столбцов ее толщину в значение 5.

Рассмотрим порядок настройки компонентов ADO на примере компонентов `ADOQuery` и `ADOConnection` (остальные компоненты ADO настраиваются аналогично). Сначала настроим компонент `ADOConnection`. Для этого выберем его свойство `ConnectionString`. После щелчка на этом свойстве в окне Инспектора объектов появляется окно Мастера настройки соединения с источником данных (рис. 33.20).

В этом окне можно выбрать вариант соединения с помощью файла связи (переключатель **Use Data Link File**) или строки соединения (переключатель **Use Connection String**). Выберем строку соединения.

Для продолжения настройки следует нажать кнопку **Build**, открывающую окно **Data Link Properties** (Свойства связи с данными). В нем нужно выбрать тип поставщика услуг (OLE DB-провайдера данных). В нашем случае мы публикуем информацию из уже использованной в других примерах демонстрационной базы данных учебника по ASP-страницам. Для нее с помощью Администратора источника данных ODBC создан псевдоним `Authors`. Поэтому выберем OLE DB-провайдер для ODBC-драйвера.

Для продолжения настройки нажмем кнопку **Next**, после чего появляется очередное окно **Data Link Properties** (Свойства связи с данными). В нем необходимо указать псевдоним для ODBC-источника данных, в нашем случае — `Authors`. Здесь же можно задать параметры соединения. В случае соединения с БД с псевдонимом `Authors` параметры соединения отсутствуют.

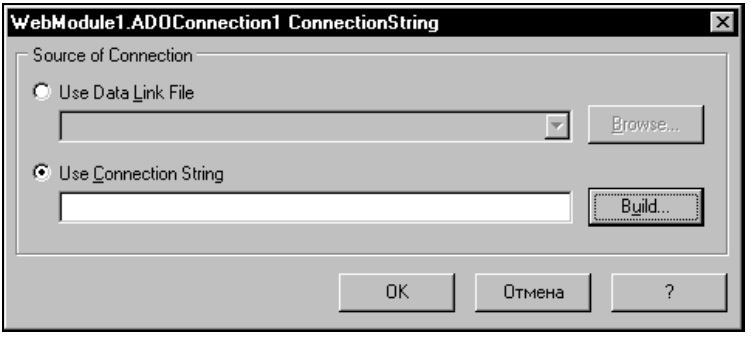


Рис. 33.20. Окно Мастера настройки соединения с источником данных

Для проверки соединения можно воспользоваться кнопкой **Test connection** (Проверить соединение). С помощью вкладки **Advanced** (Дополнительно) можно установить ре-

жимы доступа к источнику данных. Параметры соединения с ADO-источником данных можно просмотреть на вкладке **All** (Все).

После того как настройка соединения объекта `ADOConnection` завершена, в объекте `ADOQuery` для получения доступа к источнику данных достаточно установить свойство `Connection` в значение `ADOConnection1`.

Далее необходимо создать обработчик события `OnAction` для объекта `WebActionItem1`, в который нужно поместить следующий код:

```
ADOQuery1.Close;
ADOQuery1.SQL.Clear;
ADOQuery1.SQL.Add('SELECT * FROM Authors WHERE YearBorn >' +
   Request.ContentFields.Values['N1']);
ADOQuery1.Open;
Response.Content := '<HTML> <HEAD>' +
   '<TITLE> Пример публикации БД с использованием интерфейса ADO ' +
   '</TITLE> </HEAD> <BODY> <H1> Данные из БД, начиная с ' +
   Request.ContentFields.Values['N1'] + '</H1>' +
   DataSetTableProducer1.Content + '</BODY> </HTML>';
```

Здесь инструкция

```
ADOQuery1.SQL.Add('SELECT * FROM Authors WHERE YearBorn >' +
   Request.ContentFields.Values['N1']);
```

формирует строку запроса на выборку записей из БД, для которых значение поля `YearBorn` превышает величину, введенную пользователем и получаемую с помощью выражения `Request.ContentFields.Values['N1']` из поля формы с именем `N1`.

Для загрузки модуля ISAPI используется следующий HTML-документ:

```
<HTML>
<HEAD></HEAD>
<BODY>
<FORM NAME=Form1 METHOD="Post"
ACTION="http://localhost/Scripts/adow.dll">
<BR>
<H1> Эта форма позволяет посылать запрос <BR>
  на выборку данных из БД "Authors"
</H1>
<BR>
Поля для ввода года издания: <INPUT TYPE=Text NAME="N1"> <BR>
<BR>
<INPUT TYPE=Submit VALUE="Выполнить запрос">
</FORM>
</BODY>
</HTML>
```

Например, если пользователь ввел в поле `N1` значение `1954` и нажал кнопку **Выполнить запрос**, в обозревателе будет отображен HTML-документ, представленный на рис. 33.21.

На этой странице выведены записи, для которых значения поля `YearBorn` превышают `1954`.

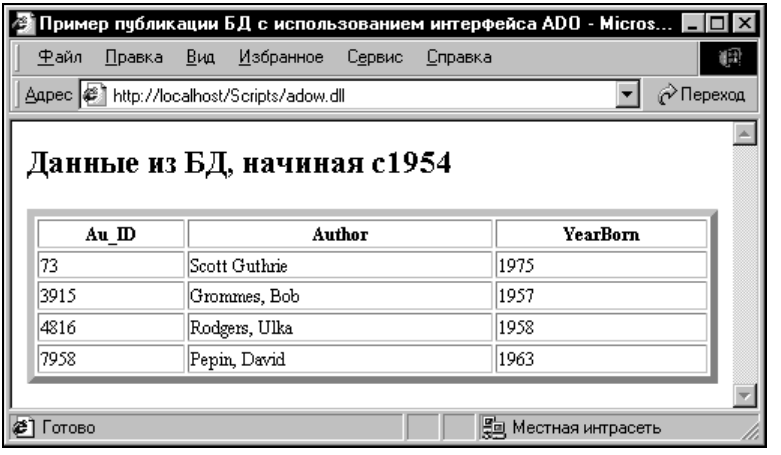
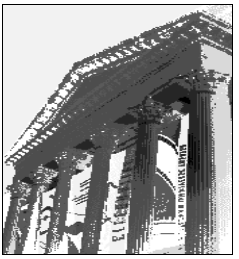


Рис. 33.21. Окно обозревателя с результатом работы модуля adow.dll



ГЛАВА 34

Работа с электронной почтой и Web-документами

Система Delphi обеспечивает возможность работы с электронной почтой (e-mail) и Web-документами в разрабатываемом приложении.

Работа с электронной почтой

Для работы с электронной почтой можно использовать функцию `ShellExecute` или интерфейс MAPI (Mail API).

Использование функции *ShellExecute*

Наиболее простым способом вызова почтового клиента, в окне которого можно подготовить (создать) отправить письмо, является применение функции `ShellExecute`.

Вот пример вызова почтового клиента:

```
// Не забудьте подключить этот модуль
uses ShellApi;
ShellExecute(Form1.Handle, Nil, 'MailTo: vladimir@mail.ru', Nil,
             Nil, SW_SHOWNORMAL);
```

В результате вызова этой функции открывается окно почтового клиента, установленного на компьютере в качестве почтовой программы по умолчанию.

Замечание

Для реализации возможности работы с электронной почтой на компьютере должна быть установлена какая-либо почтовая программа (почтовый клиент), например, Microsoft Outlook, Outlook Express, The Bat!.

Строка `MailTo:` в третьем параметре функции в приведенном выше примере указывает Windows, что необходимо запустить установленную по умолчанию почтовую программу, а необязательный адрес, идущий после нее (в нашем примере это `vladimir@mail.ru`), автоматически заносится в поле получателя.

Если необходимо автоматически заполнить поле темы письма, то оно указывается после адреса как значение параметра `Subject`. Также можно автоматически заполнить текст сообщения, для чего используется параметр `Body`.

Параметры отделяются от адреса знаком `?`, а друг от друга — знаком `&`.

Рассмотрим пример вызова почтового клиента с автоматическим заполнением адреса, темы и текста сообщения:

```
ShellExecute(Form1.Handle, nil, 'MailTo: vladimir@mail.ru' +
      '?Subject=The first message' + '&Body=It is just an example',
      Nil, Nil, SW_SHOWNORMAL);
```

Если почтовой программой по умолчанию является приложение Microsoft Outlook, и в нем задано использование Microsoft Word в качестве редактора сообщений, то в результате выполнения приведенного метода откроется окно, приведенное на рис. 34.1 (это окно является немодальным).

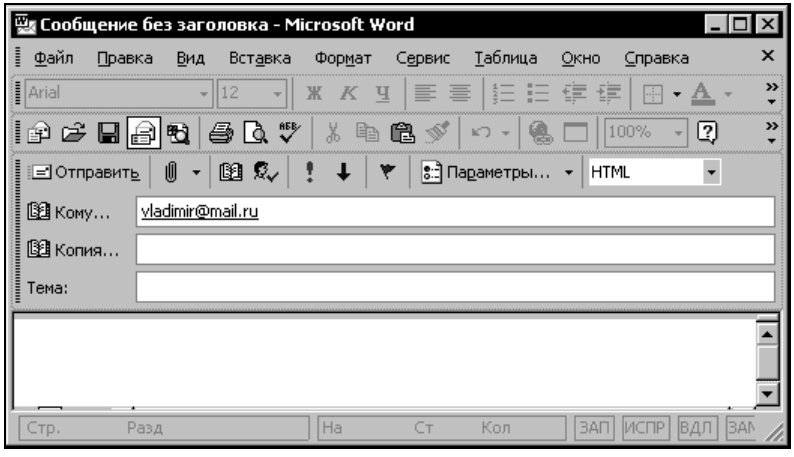


Рис. 34.1. Окно подготовки почтового сообщения

Использование интерфейса MAPI

Для работы с электронной почтой можно также использовать интерфейс MAPI, который предоставляет больше возможностей, однако программировать работу с ним труднее, чем в случае использования функции `ShellExecute`. Для работы с интерфейсом MAPI следует подключить модуль `Mapi` с помощью строки вида

```
Uses Mapi;
```

Рассмотрим пример отправки почтового сообщения из приложения с использованием MAPI. Для решения поставленной задачи в форму приложения, кроме прочих, поместим следующие компоненты: `edtReceiver`, `edtTitle`, `memMessageText`, `edtAttachment`, `OpenDialog1` и `btnInsertFile`. Поле ввода **Получатель** (`edtReceiver`) предназначено для ввода почтового адреса получателя (адресата), поле ввода **Название** (`edtTitle`) содержит название (тему) сообщения. Текст почтового сообщения набирается в многострочн-

ном редакторе `memMessageText`. Форма приложения на этапе разработки приведена на рис. 34.2.

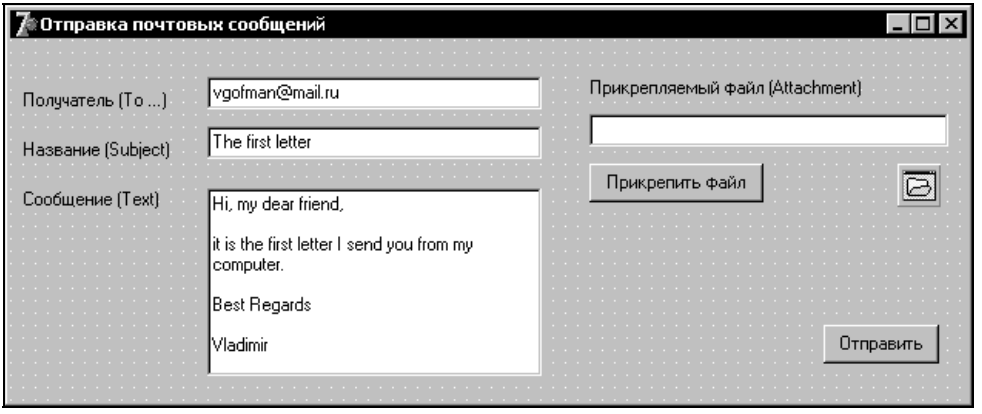


Рис. 34.2. Вид формы приложения на этапе разработки

К отправляемому сообщению можно присоединить файл, полный путь которого вводится в поле ввода **Прикрепляемый файл** (`edtAttachment`). Файл удобно выбирать, используя компонент `OpenDialog1` выбора файла. Стандартное диалоговое окно выбора файла открывается нажатием кнопки **Прикрепить файл** (`btnInsertFile`). После выбора требуемого файла его полный путь заносится в поле `edtAttachment`.

В листинге 34.1 приведен код модуля `Unit1` приложения.

Листинг 34.1. Пример отправки почтового сообщения с использованием MAPI

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
        btnSend: TButton;
    edtReceiver: TEdit;
       edtTitle: TEdit;
         Label1: TLabel;
         Label3: TLabel;
  edtAttachment: TEdit;
 memMessageText: TMemo;
         Label4: TLabel;
         Label5: TLabel;
  btnInsertFile: TButton;
    OpenDialog1: TOpenDialog;
```

```
    procedure btnSendClick(Sender: TObject);
    procedure btnInsertFileClick(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

// Этот модуль обеспечивает возможность работы с электронной почтой
uses Mapi;

{$R *.DFM}

// Подготовить и отправить почтовое сообщение
function SendEmail(Receiver, Title, MessageText,
                   AttachmentPath: string): boolean;
var MapiMessage: TMapiMessage;
         MError: Cardinal;
      Recipient: TMapiRecipDesc;
File_Attachment: TMapiFileDesc;
begin
// Подготовить информацию получателя
  FillChar(Recipient, SizeOf(Recipient), 0);
  Recipient.ulRecipClass := MAPI_TO;
  Recipient.lpszName := PChar(Receiver);

  // Подготовить информацию прикрепленного файла
  if Trim(AttachmentPath) <> '' then begin
    FillChar(File_Attachment, SizeOf(File_Attachment), 0);
    File_Attachment.nPosition := ULONG(-1);
    File_Attachment.lpszPathName := PChar(AttachmentPath);
  end;

  // Сформировать сообщение
  with MapiMessage do begin
    ulReserved := 0;
    lpszSubject := PChar(Title);
    lpszNoteText := PChar(MessageText);
    lpszMessageType := nil;
    lpszDateReceived := nil;
    lpszConversationID := nil;
    flFlags := 0;
    lpOriginator := nil;
    nRecipCount := 1;
    lpRecips := @Recipient;
```

```
    if Trim(AttachmentPath) <> '' then begin
       nFileCount := 1;
       lpFiles := @File_Attachment;
    end
    else begin
       nFileCount := 0;
       lpFiles := nil;
    end;
  end;

  // Попробовать отправить сообщение
  MError := MapiSendMail(0, 0, MapiMessage, MAPI_DIALOG, 0);
  if not (MError = SUCCESS_SUCCESS) then begin
    ShowMessage('Сообщение не было отправлено!');
    Result := False;
  end
  else Result := True;
end;

// Отправить сообщение
procedure TForm1.btnSendClick(Sender: TObject);
begin
  SendEmail(edtReceiver.Text, edtTitle.Text,
            memMessageText.Text, edtAttachment.Text);
end;

// Выбрать прикрепляемый файл
procedure TForm1.btnInsertFileClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    edtAttachment.Text := OpenDialog1.FileName;
end;
end.
```

После ввода в соответствующие поля требуемой информации письмо можно отправить, нажав кнопку **Отправить** (`btnSend`). В обработчике событий этой кнопки вызывается функция `SendEmail`, выполняющая требуемые действия по подготовке и отправке почтового сообщения (вызов почтового клиента).

Подготовка сообщения заключается в заполнении соответствующих структур типа `TMapiRecipDesc` — для получателя, типа `TMapiFileDesc` — для присоединяемого файла и типа `TMapiMessage` — для самого отправляемого сообщения.

Вызов почтового клиента выполняет функция `MapiSendMail(lhSession: LHANDLE; ulUIParam: Cardinal; var lpMessage: TMapiMessage; flFlags: FLAGS; ulReserved: Cardinal): Cardinal`.

Параметр `lhSession` содержит ссылку на сеанс, используемый для отправки данного сообщения. Если параметр имеет значение 0, то для отправки сообщения временно создается новый сеанс.

Параметр `ulUIParam` содержит ссылку на родительское окно открываемого модального диалогового окна почтовой программы. Если параметр имеет значение 0, то родителем

модального диалогового окна является приложение, из которого это диалоговое окно было вызвано.

Параметр `lpMessage` содержит отправляемое сообщение.

Параметр `flFlags` может принимать значения, важнейшие из которых:

- `MAPI_DIALOG` — указывает, что при отправке сообщения отображается диалоговое окно, в котором пользователь может ввести или изменить данные сообщения. Это окно является модальным, т. е. для продолжения работы с приложением необходимо закрыть окно сообщения. Если эта опция не установлена, то сообщение сразу отправляется получателю. При этом как минимум должен быть задан получатель сообщения;
- `MAPI_LOGON_UI` — указывает, что должно быть открыто диалоговое окно для ввода имени пользователя и пароля, если таковые установлены.

Параметр `ulReserved` зарезервирован и должен иметь значение 0.

Функция возвращает результат, основными возможными значениями которого являются следующие:

- `SUCCESS_SUCCESS` — сообщение отправлено успешно;
- `MAPI_E_ATTACHMENT_NOT_FOUND` — не найден указанный прикрепленный файл;
- `MAPI_E_ATTACHMENT_OPEN_FAILURE` — прикрепленный файл нельзя открыть;
- `MAPI_E_FAILURE` — произошла неизвестная ошибка;
- `MAPI_E_INSUFFICIENT_MEMORY` — недостаточно памяти для выполнения операции;
- `MAPI_E_TEXT_TOO_LARGE` — текст сообщения слишком велик;
- `MAPI_E_TOO_MANY_RECIPIENTS` — слишком много получателей;
- `MAPI_E_USER_ABORT` — пользователь отменил диалог.

Из приведенных вариантов следует, что если результатом функции является значение, отличное от `SUCCESS_SUCCESS`, то сообщение электронной почты не было отправлено.

Замечание

Если на компьютере установлен почтовый клиент, отличный от Microsoft Outlook, то отправка сообщений с использованием модуля MAPI может выполняться некорректно.

Перед попыткой отправить сообщение с помощью кнопки **Отправить** почтовый клиент, например Microsoft Outlook, должен быть запущен. В противном случае будет выдано предупреждение "Your message has not been sent!" (Ваше сообщение не было отправлено!).

Работа с Web-документами

Система Delphi обеспечивает возможность работы с Web-документами в разрабатываемом приложении так же, как и с помощью Web-обозревателя (браузера) MS Internet Explorer. В качестве примеров прикладных задач работы с Web-документами с помощью приложений Delphi можно указать задачи: обработки статистики, например, данных о котировках акций на биржах, поиска информации в документах и др.

Для работы с Web-документами в Delphi предназначен компонент WebBrowser, который размещен на странице **Internet** Палитры компонентов.

Характеристика компонента *WebBrowser*

Рассматриваемый компонент WebBrowser типа TWebBrowser позволяет работать с Web-документами, выполняя различные операции: загрузки, просмотра содержимого, сохранения, печати, поиска и форматирования текста и др. Он использует функциональность компонента WebBrowser, входящего в состав обозревателя Microsoft Internet Explorer версий 4 и старше. Важным представляется то обстоятельство, что с помощью компонента WebBrowser в приложении Delphi можно выполнить отображение информации любого типа, которая представима с помощью обозревателя Microsoft Internet Explorer.

Основным *методом* компонента WebBrowser является перегружаемый метод Navigate, который обеспечивает управление навигацией к указанному ресурсу. Метод имеет несколько прототипов, например:

```
procedure Navigate(const URL: WideString); overload;
procedure Navigate(const URL: WideString; var Flags: OleVariant;
var TargetFrameName: OleVariant; var PostData: OleVariant;
                    var Headers: OleVariant); overload;
```

Параметры метода:

- URL — задает адрес открываемого документа;
- Flags — битовая маска из нескольких флагов для настройки параметров работы с Web-документом. При формировании маски используются следующие константы:
  - navOpenInNewWindow, 1 — документ открывается в новом окне обозревателя по умолчанию;
  - navNoHistory, 2 — адрес документа не заносится в список **History**;
  - navNoReadFromCache, 4 — запрашиваемая страница загружается с сервера, а не из кэша;
  - navNoWriteToCache, 8 — открываемая страница не записывается в кэш;
  - navAllowAutosearch, 16 — разрешается использовать автоматический поиск домена;
- TargetFrameName — определяет имя фрейма, в который требуется загрузить страницу. При задании значения NULL страница загружается в текущее окно;
- PostData — определяет данные для передачи на сервер методом HTTP POST. При задании значения NULL для передачи данных будет использоваться метод HTTP GET;
- Headers — задает заголовок HTTP для передачи на сервер, в котором указываются требуемые действия, типы данных и т. п.

Например, для того чтобы в компоненте WebBrowser отобразить страницу HTML, адрес которой задается в простом редакторе с именем URL_Edit, достаточно в обработчике нажатия кнопки BtNavigate записать код:

```
procedure TForm1.BtNavigateClick(Sender: TObject);
var
  URL, Flags, TargetFrameName, PostData, Headers: Olevariant;
begin
  try  URL := URL_Edit.Text;
    Flags := 0;
    TargetFrameName := 0;
    Postdata := 0;
    Headers := 0;
    WebBrowser1.Navigate2(URL, Flags, TargetFrameName, PostData, Headers);
  except
  end;
end;
```

Для управления просмотром Web-документов при использовании компонента `WebBrowser` предназначены следующие *методы*:

- `procedure GoBack;` — переход назад;
- `procedure GoForward;` — переход вперед;
- `procedure GoHome;` — переход к домашней странице;
- `procedure GoSearch;` — выполнение поиска;
- `procedure Refresh;` — обновление текущей страницы;
- `procedure Stop;` — остановка загрузки страницы;
- `procedure Quit;` — выход из обозревателя.

Отметим, что Web-документ состоит из одного или нескольких фреймов. Каждый из фреймов представляется с помощью компонента `WebBrowser`, который входит в состав фрейма более высокого уровня. Фрейм самого верхнего уровня и есть тот самый элемент управления, который представляется созданным нами компонентом `WebBrowser` типа `TWebBrowser`. Этот компонент существует постоянно, а компоненты `WebBrowser` более низкого уровня создаются и уничтожаются динамически в зависимости от выполняемого варианта навигации — к многофреймовым или к однофреймовым документам.

К фрейму верхнего уровня можно получить доступ через методы и свойства соответствующего компонента `WebBrowser` и через соответствующие интерфейсные ссылки. К методам `WebBrowser` подчиненных фреймов — только через интерфейсные ссылки. Интерфейсную ссылку на `WebBrowser` верхнего уровня можно получить через его свойство `ControlInterface` или `DefaultInterface`. Получить интерфейсные ссылки на `WebBrowser` нижнего уровня можно при помощи некоторых обработчиков событий, которые сопровождают процесс навигации.

К числу основных *событий* рассматриваемого нами компонента можно отнести следующие события:

- `OnDownloadBegin` — возникает перед началом навигации по заданному адресу URL;
- `OnDownloadComplete` — возникает в момент окончания загрузки Web-страницы;
- `OnBeforeNavigate2` — возникает перед началом навигации;

- `OnNewWindow2` — возникает перед открытием нового окна обозревателя;
- `OnNavigateComplete2` — возникает при завершении навигации по заданному адресу;
- `OnDocumentComplete` — возникает при завершении загрузки каждого фрейма документа.

Для примера приведем код, иллюстрирующий как можно использовать события `OnNavigateComplete2` и `OnDocumentComplete` для отслеживания момента полного завершения загрузки документа, а не только отдельных его фреймов:

```
var
   CurDisp: IDispatch;
  // Глобальная переменная состояния навигации
. . .
procedure TForm1.WebBrowser1NavigateComplete2(Sender: TObject;
  const pDisp: IDispatch; var URL: OleVariant);
begin
  if CurDisp = nil then
    CurDisp := pDisp;
    // Сохраняем для сравнения
end;

procedure TForm1.WebBrowser1DocumentComplete(Sender: TObject;
  const pDisp: IDispatch; var URL: OleVariant);
begin
  if (pDisp = CurDisp) then
  begin
    Beep;
    CurDisp := nil;
    // Очистка глобальной переменной
  end;
end;
```

При завершении загрузки документа в обработчике `WebBrowser1DocumentComplete` выдается звуковой сигнал и осуществляется очистка глобальной переменной состояния навигации.

Замечание

Чтобы отключить сообщения об ошибках, возникающих при выполнении сценариев в ходе загрузки или навигации по Web-документу, свойству `Silent` компонента `WebBrowser` с помощью Инспектора объектов следует установить значение `True`. Сделать это можно и программно, разместив непосредственно перед оператором выполнения навигации следующий код:

```
WebBrowser1.Silent:=True;
```

При использовании компонента `WebBrowser` должны быть подключены модули `MSHTML` и `SHDocVw` путем указания их в предложении `Uses`. В этих модулях находятся описания интерфейсов, позволяющих организовать взаимодействие с рассматриваемым компонентом.

Управление с помощью процедуры *ExecWB*

Для управления просмотром Web-документов при использовании компонента `WebBrowser` и выполнения различных вспомогательных действий (печати, просмотра,

редактирования, сохранения и др.) предназначена перегружаемая процедура с прототипом вида:

```
procedure ExecWB(cmdID: OLECMDID; cmdexecopt: OLECMDEXECOPT); overload;
```

Параметры метода:

- `CmdID` — задает команду, которую нужно выполнить. Параметр может принимать свыше трех десятков значений, в том числе: `OLECMDID_OPEN`, `OLECMDID_NEW`, `OLECMDID_SAVE`, `OLECMDID_SAVEAS`, `OLECMDID_PRINT`, `OLECMDID_CUT`, `OLECMDID_COPY`, `OLECMDID_UNDO`, `OLECMDID_REDO`, `OLECMDID_SELECTALL`, `OLECMDID_REFRESH`, `OLECMDID_STOP`, `OLECMDID_FIND`, `OLECMDID_DELETE`. Ряд из них дублируется указанными выше процедурами, например `Refresh` и др. Остальные значения позволяют задать выполнение важных команд;
- `Cmdexecopt` — уточняет способ выполнения команды. Может принимать значения: `OLECMDEXECOPT_DODEFAULT` (команда выполняется, как принято по умолчанию); `OLECMDEXECOPT_PROMPTUSER` (предварительно выводится диалоговое окно настроек); `OLECMDEXECOPT_DONTPROMPTUSER` (команда выполняется без дополнительных вопросов); `OLECMDEXECOPT_SHOWHELP` (выводится справка по заданному действию без выполнения самой команды).

Например, при выполнении команды

```
WebBrowser1.ExecWB(OLECMDID_SAVEAS,OLECMDEXECOPT_PROMPTUSER);
```

открывается стандартный диалог сохранения Web-документа. При этом можно выбрать вариант сохранения: Web-страницу полностью, Web-архив, Web-страницу (только HTML), текстовый файл.

Работа в режиме HTML-редактора

При необходимости компонент `WebBrowser` можно использовать в качестве редактора HTML-документа[1]. Для этого нужно перевести указанный компонент в режим редактирования, например, с помощью следующего кода:

```
var
Disp: IDispatch;
Editor: IHTMLDocument2;
. . .
procedure TForm1.DesignModeClick(Sender: TObject);
var  CurrentWB: IWebBrowser;
begin  CurrentWB := Disp as IWebBrowser;
  Editor:=(CurrentWB.Document as IHTMLDocument2);
  Editor.DesignMode := 'On';
end;
```

Здесь приведены объявления двух глобальных переменных и обработчик `DesignModeClick` нажатия кнопки, в котором для текущего обозревателя на основе компонента `WebBrowser` (задается с помощью переменной `Editor`) устанавливается режим редактирования.

---

[1] В приведенных примерах нами использованы предложения Дениса Боднара, высказанные им на форуме (**www.samum2000.narod.ru**, опубликовано 27.10.2006).

После перевода обозревателя в режим редактирования можно визуально и программно выполнять разнообразные действия над содержимым HTML-документа. Например, с помощью мыши или клавиатуры можно выделить некоторый фрагмент документа. Затем с помощью обработчика вида:

```
procedure TForm1.SpBtBoldClick(Sender: TObject);
var
Range: IHTMLTxtRange;
begin
  Range:=(Editor.selection.createRange as IHTMLTxtRange);
  Range.execCommand('bold',false,emptyparam)
end;
```

можно выполнить переключение начертания шрифта выделенного текста на полужирное и обратно.

Кроме того, в режиме редактирования можно выполнить копирование выделенного фрагмента в редактор `Edit` или компонент `Memo`, например, как это делается в следующем обработчике:

```
procedure TForm1.BtGetHTMLClick(Sender: TObject);
var  HText: WideString;
     Range: IHTMLTxtRange;
begin
  Range:=(Editor.selection.createRange as IHTMLTxtRange);
  HText:=Range.Get_htmlText;
  Edit2.Text:=  HText;
  Memo1.Lines[0]:=HText;
end;
```

При этом в компоненте-приемнике (в примере это компоненты `Edit` или `Memo`) будет передана копия выделенного фрагмента HTML-документа вместе с тегами, если таковые окажутся в выделенном фрагменте.

Можно также выполнить обратную передачу текста, скажем, из простого редактора в Web-обозреватель, например, так:

```
procedure TForm1.BtSetHTMLClick(Sender: TObject);
var  HText: WideString;
     Range: IHTMLTxtRange;
begin
  HText := Edit2.Text;
  Range:=(Editor.selection.createRange as IHTMLTxtRange);
  Range.Set_Text(HText)
end;
```

Как видим, здесь в Web-обозреватель передается (вставляется) текст из простого редактора `Edit2`.

Пример формы приложения

Для примера рассмотрим форму приложения (рис. 34.3), с помощью которой можно работать с обозревателем Интернета на основе компонента `WebBrowser`, выполняя

основные действия при работе с Web-документами: открытие документа, переход по адресу, переходы вперед и назад по истории навигации, обновление документа, поиск и выделение текста в документе, сохранение и печать документа. По существу здесь, на наш взгляд, приводится один из основных вариантов использования `WebBrowser` компонента в приложении Delphi — поиск нужного фрагмента информации и сохранение его в нужном месте для последующего использования.

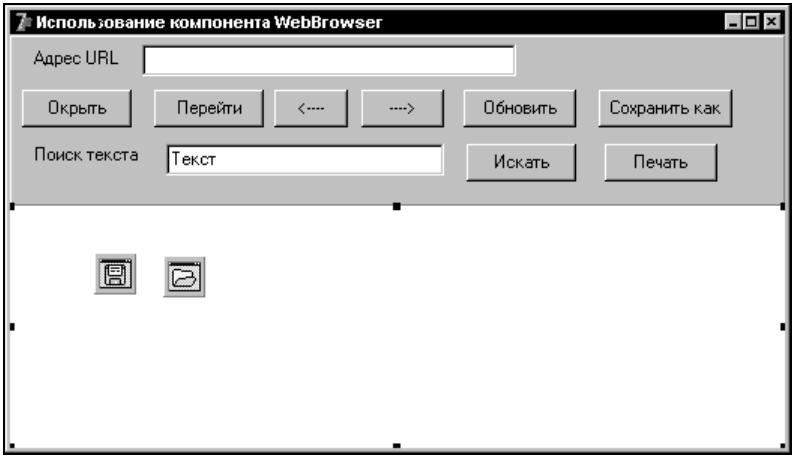


Рис. 34.3. Вид формы приложения для работы с Web-документами

В листинге 34.2 приведен код модуля формы приложения, реализующего поставленную задачу.

Листинг 34.2. Использование компонента `WebBrowser`

```
unit UnitWebBrowser;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, OleCtrls, SHDocVw, MSHTML;

type
  TDoerOneDoc = procedure(iDoc: IHtmlDocument2);
  TForm1 = class(TForm)
    WebBrowser1: TWebBrowser;
         Panel1: TPanel;
        EditURL: TEdit;
      BtGoToURL: TButton;
     BtOpenFile: TButton;
       BtGoBack: TButton;
    BtGoForward: TButton;
      BtRefresh: TButton;
       BtSaveAs: TButton;
        BtPrint: TButton;
```

```
    OpenDialog1: TOpenDialog;
     BtFindText: TButton;
       LabelURL: TLabel;
  LabelFindText: TLabel;
   EditFindText: TEdit;
    SaveDialog1: TSaveDialog;
    procedure BtGoBackClick(Sender: TObject);
    procedure BtGoForwardClick(Sender: TObject);
    procedure BtRefreshClick(Sender: TObject);
    procedure BtGoToURLClick(Sender: TObject);
    procedure BtOpenFileClick(Sender: TObject);
    procedure BtPrintClick(Sender: TObject);
    procedure BtFindTextClick(Sender: TObject);
    procedure BtSaveAsClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.BtGoBackClick(Sender: TObject);
begin
  WebBrowser1.GoBack;
  // Переход по истории навигации назад
end;

procedure TForm1.BtGoForwardClick(Sender: TObject);
begin
  WebBrowser1.GoForward;
  // Переход по истории навигации вперед
end;

procedure TForm1.BtRefreshClick(Sender: TObject);
begin
  WebBrowser1.Refresh;
  // Обновление страницы
end;

procedure TForm1.BtGoToURLClick(Sender: TObject);
begin
  if EditURL.Text<>'' then
    WebBrowser1.Navigate(EditURL.text);
    // Переход к адресу URL
end;
```

```
procedure TForm1.BtOpenFileClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    EditURL.Text:=Opendialog1.FileName;
    // Имя файла после открытия отображается в редакторе
    WebBrowser1.Navigate(Opendialog1.FileName);
    // Диалог открытия файла на локальном диске
  end;
end;

procedure TForm1.BtPrintClick(Sender: TObject);
begin
  WebBrowser1.ExecWB(OLECMDID_PRINT,OLECMDEXECOPT_PROMPTUSER);
  // Диалог печати файла документа
end;

procedure TForm1.BtFindTextClick(Sender: TObject);
var
  PageContents: IHTMLTxtRange;
  s: string;
begin
  if not(EditFindText.Text='') then
  begin
    if not(assigned(WebBrowser1.ControlInterface.Document)) then exit;
    if not(WebBrowser1.Busy) then
    begin
      s:=EditFindText.Text;
      PageContents:=((WebBrowser1.Document as IHTMLDocument2).body
             as IHTMLBodyElement).createTextRange;
      // Подготовка страницы для поиска текста
      while PageContents.findText(s,1,0) do
      // Поиск заданного текста на всей странице
      begin
        PageContents.pasteHTML('<font color=red><b>'+
              PageContents.htmlText+'</font></b>');
        // Выделение очередного найденного текста
        PageContents.scrollIntoView(True);
        // Переход к последнему найденному тексту
      end;
    end;
  end;
end;

procedure TForm1.BtSaveAsClick(Sender: TObject);
begin
  WebBrowser1.ExecWB(OLECMDID_SAVEAS,OLECMDEXECOPT_DODEFAULT);
  // Вызывается диалог сохранения Web-документа
end;
end.
```

Все визуальные компоненты (элементы управления) приложения, за исключением компонента `WebBrowser`, для удобства размещены на панели (компонент `Panel1` типа `TPanel`).

В обработчике `BtOpenFileClick` нажатия кнопки **Открыть** выполняется открытие диалога выбора файла Web-документа и его открытие в окне компонента с помощью метода `Navigate`. При этом предыдущая строка кода обеспечивает отображение адреса открытого файла документа.

Замечание

У компонента `WebBrowser` соответствующее свойство `AddressBar` не используется, поскольку, в отличие от Internet Explorer, он не имеет адресной панели.

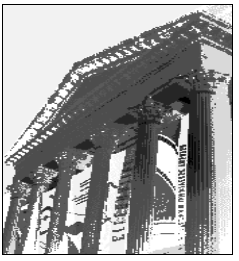
В обработчике `BtGoToURLClick` нажатия кнопки **Перейти** с помощью метода `Navigate` выполняется переход по адресу URL, указанному в качестве свойства `Text` простого редактора `EditURL`.

В обработчиках `BtGoBackClick` и `BtGoForwardClick` нажатия кнопок со стрелками ← и → выполняются переходы по истории навигации назад и вперед соответственно.

В обработчике `BtFindTextClick` нажатия кнопки **Искать** выполняется поиск текста[1], размещенного в свойстве `Text` простого редактора `EditFindText`. При этом используется свойство `ControlInterface` компонента `WebBrowser`, которое обеспечивает доступ к интерфейсу этого компонента. Переменная `PageContents` типа `IHTMLTxtRange` содержит HTML-страницу, в которой выполняется поиск вхождений строки текста с помощью метода `findText`. Для каждого из найденных вхождений строки текста выполняется его шрифтовое выделение (красный цвет шрифта и полужирное начертание).

[1] Пример процедуры поиска текста приводится в очень удачной, на наш взгляд, книге: Шкрыль А. Delphi. Народные советы. — СПб.: БХВ-Петербург, 2007.

ГЛАВА 35



Характеристика Web-служб

Web-службы, получившие дальнейшее развитие в Delphi 7, предназначены для обеспечения универсальности доступа к данным Интернета.

Основные понятия

Для разработки и использования Web-служб в Delphi используются широко распространенный протокол HTTP и ставший своего рода стандартом обмена данными язык XML. Тем самым обеспечивается универсальная двусторонняя связь между клиентом и сервером. При этом от клиента не требуется использование определенной платформы или языка программирования.

Web-службы представляют собой модульные приложения, которые могут быть опубликованы и вызваны в Интернете. Возможен программный доступ к ним со стороны клиентских приложений.

Работа с Web-службами ведется с помощью протокола SOAP (Simple Object Access Protocol, простой протокол доступа к объектам), являющегося стандартом легковесного протокола обмена информацией в децентрализованной распределенной среде. Этот протокол использует язык XML для кодирования удаленных вызовов процедур и обычно применяет HTTP в качестве коммуникационного протокола.

Компоненты, поддерживающие Web-службы в Delphi, ориентированы на использование протоколов SOAP и HTTP, но основа их является достаточно общей и позволяет использование других протоколов кодирования и коммуникации.

С помощью средств Delphi 7 можно создавать приложения-серверы Web-служб, а специальные компоненты и мастера позволяют создавать клиенты для Web-служб, которые используют SOAP-кодирование.

Приложения Web-служб публикуют информацию о том, через какой интерфейс они доступны и как осуществлять их вызов, с помощью документов WSDL (Web Service Definition Language, язык определения Web-службы). На стороне сервера приложение может опубликовать документ WSDL, описывающий Web-службу. На стороне клиента мастер или утилита командной строки позволяют импортировать опубликованный до-

кумент WSDL, обеспечивая клиента определениями интерфейса и требуемой для связи информацией. Если уже имеется документ WSDL, описывающий предполагаемую к выполнению Web-службу, то можно сгенерировать код на стороне сервера, как и при импортировании документа WSDL.

При создании клиентского приложения, использующего Web-службу, требуется описание выполняемых ею действий и сведений, передаваемых Web-службе при обращении к ней. К примеру, клиенту нужно знать предоставляемые Web-службой методы, их параметры и используемые протоколы. В определенном смысле названная информация аналогична информации из библиотеки типов, используемой стандартным компонентом COM. Недостатком использования библиотеки типов Microsoft COM является то, что ее клиентами могут быть только приложения, работающие под управлением операционных систем фирмы Microsoft.

Использование Web-служб устраняет названный недостаток, при этом их услугами могут пользоваться клиентские приложения, являющиеся разработками любых фирм. Важно только, чтобы при обращении к серверу Web-службы использовались указанные выше протоколы и кодирование.

Документ WSDL

Как отмечалось, описывающий Web-службу документ WSDL кодируется на языке XML. Фрагменты кода XML WSDL-документа для примера приводятся в листинге 35.1.

Листинг 35.1. Фрагменты XML-кода WSDL-документа

```
// ************************************************************************ //
// The types declared in this file were generated from data read from
// the WSDL File described below:
// WSDL: https://www.software-events.net/mSiteBuilderServices/
//   events.asmx?WSDL
// >Import: https://www.software-events.net/mSiteBuilderServices/
//   events.asmx?schema=sitesDataSet
// >Import: https://www.software-events.net/mSiteBuilderServices/
//   events.asmx?schema=eventGroupDataSet
...
// >Import: https://www.software-events.net/mSiteBuilderServices/
//   events.asmx?schema=revocationListDataSet
// Encoding : utf-8
// Version  : 1.0
// (13.11.2002 15:37:15 — 1.33.2.5)
//************************************************************************//
unit events;
interface
uses InvokeRegistry, SOAPHTTPClient, Types, XSBuiltIns;
const
  sUDDIOperator   = 'http://test.uddi.microsoft.com/inquire';
  sUDDIBindingKey = 'f1471305-b656-405b-940c-688bce0f6d12';
```

```
type
//**********************************************************************//
// The following types, referred to in the WSDL document are not
// being represented in this file. They are either aliases[@] of
// other types represented or were referred to but never[!] declared
// in the document. The types from the latter category typically map
// to predefined/known XML or Borland types; however, they could also
// indicate incorrect WSDL documents that failed to declare or import
// a schema type.
//**********************************************************************//
  // !:boolean         — "http://www.w3.org/2001/XMLSchema"
  // !:int             — "http://www.w3.org/2001/XMLSchema"
  // !:string          — "http://www.w3.org/2001/XMLSchema"
  // !:dateTime        — "http://www.w3.org/2001/XMLSchema"
  // !:decimal         — "http://www.w3.org/2001/XMLSchema"
  // !:short           — "http://www.w3.org/2001/XMLSchema"

  loginState = class;
                    { "http://www.software-events.net/mSiteBuilder" }
  GetMySitesResult  = class;
                    { "http://www.software-events.net/mSiteBuilder" }
  GetMyGroupsResult = class;
                    { "http://www.software-events.net/mSiteBuilder" }
  GetMyVenuesResult = class;
                    { "http://www.software-events.net/mSiteBuilder" }
...
{ "http://www.software-events.net/mSiteBuilder" }
  statusType = (All, True, False);
  { "http://www.software-events.net/mSiteBuilder" }
  accessType = (All2, Owned, Shared, ReadOnly, Public);
//**********************************************************************//
  // Namespace : http://www.software-events.net/mSiteBuilder
//**********************************************************************//
  loginState = class(TRemotable)
  private
    FIsAuthenticated: Boolean;
    FLoggedInAs: WideString;
  published
    property IsAuthenticated: Boolean read FIsAuthenticated
                                          write FIsAuthenticated;
    property LoggedInAs: WideString read FLoggedInAs
                                          write FLoggedInAs;
  end;
...
//**********************************************************************//
// Namespace : http://www.software-events.net/mSiteBuilder
// soapAction: http://www.software-events.net/mSiteBuilder/
//   %operationName%
// transport : http://schemas.xmlsoap.org/soap/http
// style     : document
// binding   : mSiteBuilderSoap
```

```
// service   : mSiteBuilder
// port      : mSiteBuilderSoap
// URL       : https://www.software-events.net/mSiteBuilderServices/
//   events.asmx
//************************************************************//
  mSiteBuilderSoap = interface(IInvokable)
  ['{8B2B09D4-39E0-ECD4-F2A2-7243249A2B69}']
    function  Login(const mPUID: WideString): Boolean; stdcall;
    ...
    function  AddMyEventToGroup(const groupID: Integer;
                      const eventID: Integer; const
                      accessLevel: accessType): Boolean; stdcall;
  end;

function GetmSiteBuilderSoap(UseWSDL: Boolean=System.False;
       Addr: string=''; HTTPRIO: THTTPRIO = nil): mSiteBuilderSoap;

implementation

function GetmSiteBuilderSoap(UseWSDL: Boolean; Addr: string;
       HTTPRIO: THTTPRIO): mSiteBuilderSoap;
const
  defWSDL =
  'https://www.software-events.net/mSiteBuilderServices/events.asmx?WSDL';
  defURL  =
  'https://www.software-events.net/mSiteBuilderServices/events.asmx';
  defSvc  = 'mSiteBuilder';
  defPrt  = 'mSiteBuilderSoap';
var
  RIO: THTTPRIO;
begin
  Result := Nil;
  if (Addr = '') then
  begin
    if UseWSDL then
      Addr := defWSDL
    else
      Addr := defURL;
  end;
  if HTTPRIO = nil then
    RIO := THTTPRIO.Create(nil)
  else
    RIO := HTTPRIO;
  try
    Result := (RIO as mSiteBuilderSoap);
    if UseWSDL then
    begin
      RIO.WSDLLocation := Addr;
      RIO.Service := defSvc;
      RIO.Port := defPrt;
    end else
      RIO.URL := Addr;
```

```
  finally
    if (Result = Nil) and (HTTPRIO = Nil) then
      RIO.Free;
  end;
end;

initialization
  InvRegistry.RegisterInterface(TypeInfo(mSiteBuilderSoap),
       'http://www.software-events.net/mSiteBuilder', 'utf-8');
  ...
  RemClassRegistry.RegisterXSInfo(TypeInfo(statusType),
       'http://www.software-events.net/mSiteBuilder', 'statusType');
  RemClassRegistry.RegisterXSInfo(TypeInfo(accessType),
       'http://www.software-events.net/mSiteBuilder', 'accessType');

  RemClassRegistry.RegisterXSClass(GetRevocationListResult,
       'http://www.software-events.net/mSiteBuilder',
       'GetRevocationListResult');

end.
```

Как видно из приведенного кода, документ WSDL включает объявления классов, прототипы функций, объявления вызываемого интерфейса и другие элементы. В разделе `implementation` содержится описание функции с именем `Get<имя службы>`, возвращающей вызываемый интерфейс. В приведенном листинге функция имеет имя `GetmSiteBuilderSoap`.

Дадим для объекта `RIO` короткие пояснения по поводу его свойств `WSDLLocation`, `Service`, `Port` типа `string`, которые используются в функции `GetmSiteBuilderSoap` для организации связи с Web-службой. Свойство `WSDLLocation` служит для указания адреса местоположения документа WSDL, свойства `Service` и `Port` содержат доступные по этому адресу наборы служб и портов соответственно. С помощью порта мы получаем доступ к конкретной Web-службе.

Вызываемый интерфейс

Серверы, поддерживающие Web-службы, построены с использованием *вызываемых интерфейсов*, которые скомпилированы для включения информации о типе времени выполнения (RunTime Type Information, RTTI). На стороне сервера эта информация используется при интерпретации поступающих вызовов методов от клиента, так чтобы они могли быть корректно введены. На стороне клиента информация о типе времени выполнения используется для того, чтобы динамически сгенерировать таблицу методов для подготовки вызова методов интерфейса.

Для создания вызываемого интерфейса требуется скомпилировать интерфейс с параметром компиляции `{$M+}`. Потомок любого вызываемого интерфейса также является вызываемым. Однако если вызываемый интерфейс происходит от другого интерфейса, не являющегося вызываемым, то Web-служба может использовать только методы, определенные в вызываемом интерфейсе и его потомках. Методы, наследуемые от невы-

зываемых предков, не компилируются с информацией о типе и не могут использоваться как часть Web-службы.

При использовании вызываемого интерфейса автоматически поддерживается работа со следующими встроенными скалярными типами данных: `Boolean`, `ByteBool`, `WordBool`, `LongBool`, `Char`, `Byte`, `ShortInt`, `SmallInt`, `Word`, `Integer`, `Cardinal`, `LongInt`, `Int64`, `Single`, `Double`, `Extended`, `String`, `WideString`, `Currency`, `TDateTime`, `Variant`. При работе с ними от программиста не требуются никакие дополнительные действия. При необходимости использовать другие типы данных нужно выполнить их регистрацию с помощью реестра удаляемых типов.

Страница *WebServices* Палитры компонентов

На странице **WebServices** Палитры компонентов (рис. 35.1) содержатся следующие компоненты, обеспечивающие работу с Web-службами:

- `HTTPRIO` — использует сообщения HTTP для вызова объекта удаленного интерфейса с помощью протокола SOAP;
- `HTTPReqResp` — выполняет обращение к методу на вызываемом интерфейсе путем посылки сообщения SOAP на сервер;
- `OPToSoapDomConvert` — управляет загрузкой и выгрузкой методов вызова SOAP;
- `SoapConnection` — используется в клиентской части приложения распределенной базы данных для установления и обслуживания соединения между клиентом и удаленным сервером приложения, который выполнен как Web-служба;
- `HTTPSoapDispatcher` — отвечает на сообщения SOAP путем пересылки их программе вызова для интерпретации;
- `WSDLHTMLPublish` — выполняет публикацию списка документов WSDL, описывающих приложение Web-службы;
- `HTTPSoapPascalInvoker` — интерпретирует сообщение запроса SOAP и выполняет соответствующий вызываемый интерфейс.

Рис. 35.1. Страница **WebServices** Палитры компонентов

Рассмотрим кратко особенности использования перечисленных компонентов.

Компонент `HTTPRIO` служит для генерации статически связанных обращений к вызываемым интерфейсам на удаленном приложении Web-службы. Когда приложение назначает зарегистрированному вызываемому интерфейсу компонент `HTTPRIO`, он динамически генерирует в основной памяти таблицу методов, обеспечивающую реализацию этого вызываемого интерфейса. При этом компонент `HTTPRIO` выполняет методы в этой таблице методов путем кодирования вызова метода как запроса SOAP и посылки сообщения с запросом HTTP приложению Web-службы. Кроме того, он распаковывает

результирующее сообщение с ответом HTTP, чтобы получить возвращаемое значение и любые выходные параметры или чтобы вызвать исключение, если запрос вызвал исключение на сервере.

Опубликованные свойства компонента `HTTPRIO` используют, чтобы указать, как связываться с приложением Web-службы. Есть два способа указать, где размещено приложение сервера:

- использовать свойство `URL`, чтобы определить URL, по которому расположено приложение сервера;
- для динамического использования информации о соединении из документа WSDL можно установить свойство `WSDLLocation` в нужное значение. После этого надо выбрать значения для свойств `Service` и `Port` в раскрывающемся списке в окне Инспектора объектов, чтобы определить используемые связи.

Прежде чем компонент `HTTPRIO` сможет генерировать таблицу методов для вызываемого интерфейса, последний должен быть зарегистрирован в реестре вызовов. Для этого используются глобальная функция `InvRegistry` и ее метод `RegisterInterface`.

Компонент `HTTPReqResp` управляет основанной на протоколе HTTP связью с провайдером Web-службы по требованию компонента `HTTPRIO`. Компонент `HTTPRIO` использует этот компонент, чтобы установить соединение с провайдером Web-службы и выполнить следующие две задачи: выдать запрос `GET` для получения информации из документа WSDL; выдать запрос `POST`, чтобы передать метод вызова серверу для выполнения и получить обратно результаты.

При этом целевую Web-службу HTTP-сообщений можно задать напрямую с помощью свойств `SoapAction` и `URL` или использовать свойство `SoapAction` в документе WSDL, как определено свойством `WSDLView`. Дополнительные свойства обеспечивают детали, включенные в заголовок HTTP-запроса сообщения.

Компонент `HTTPReqResp` в Windows для установления соединения с сервером использует динамическую библиотеку WinInet. Отметим, что библиотека wininet.dll должна быть установлена в клиентской системе, в Windows она обычно размещается в папке System, если установлен Internet Explorer 3 и выше. В Windows 2000 эта библиотека размещена в папке System32.

Компонент `OPToSoapDomConvert` представляет собой реализацию интерфейса `IOPConvert`, управляющего загрузкой и выгрузкой методов вызова SOAP. Он использует парсеры (синтаксические анализаторы) DOM для проверки правильности кодирования методов вызова SOAP и их результатов.

Обычно приложения не используют компонент `OPToSoapDomConvert` напрямую. Вместо этого он создается и используется компонентом `HTTPRIO` в клиентских приложениях и компонентами `HTTPSoapPascalInvoker` или `THTTPSoapCppInvoker` — в серверных приложениях.

Компонент `OPToSoapDomConvert` выполняет преобразование между кодированием SOAP вызова метода или его результатами и содержимым (контекстом) вызова. Если вызов метода порождает исключение, то компонент `OPToSoapDomConvert` генерирует ошибочный пакет, так что клиент может вызвать соответствующее исключение удаленно.

Компонент `SoapConnection` позволяет:

- устанавливать начальное соединение с удаленным сервером приложения с помощью SOAP;
- получать интерфейс для сервера приложения;
- получать список провайдеров на сервере приложения;
- отменять соединение с удаленным сервером приложения.

Для использования компонента `SoapConnection` в Windows, как и в случае компонента `HTTPReqResp`, должна быть установлена библиотека wininet.dll в клиентской системе.

Компонент `SoapConnection` использует внутренний объект `HTTPRIO`, чтобы получить интерфейс от приложения Web-службы. Им может быть интерфейс `IAppServer` или `IAppServerSOAP`, или один из их потомков в зависимости от значения свойства `UseSOAPAdapter`. Рекомендуется использовать интерфейс `IAppServerSOAP` как наиболее адаптированный для протокола SOAP.

Некоторые приложения сервера, например, создаваемые с помощью Delphi 7 или Kylix 2, поддерживают только интерфейс `IAppServer`. В этом случае свойство `UseSOAPAdapter` компонента `SoapConnection` следует установить в значение `False`.

Независимо от используемого для связи с сервером приложения интерфейса компонент `SoapConnection` делает доступной информацию клиентским наборам данных в приложении клиента посредством обеспечения интерфейса `IAppServer`. Клиентские наборы данных используют интерфейс `IAppServer` из компонента `SoapConnection` для связи с провайдерами на сервере приложения или, наоборот, для вызова интерфейса модуля данных сервера приложения.

Компонент `HTTPSoapDispatcher` используют в приложении, публикующем Web-службу с применением протокола SOAP. Этот компонент отвечает на все обращения к вызываемым интерфейсам, зарегистрированным в приложении. Регистрация их выполняется с помощью метода `RegisterInterface` реестра вызовов `InvRegistry`.

Рассматриваемый компонент `HTTPSoapDispatcher` действует как диспетчер, принимая поступающие сообщения и направляя их другому компоненту, который управляет задачей их интерпретации и обработки. Названный компонент идентифицируется с помощью свойства `Dispatcher`. Отметим, что диспетчер представляет собой интерфейс вызывающего компонента, который интерпретирует сообщение SOAP, идентифицирует вызываемый интерфейс, являющийся целью вызова, выполняет вызов и формирует содержимое ответного сообщения.

Компонент `HTTPSoapDispatcher` автоматически регистрирует себя вместе с модулем или диспетчером Web как автодиспетчируемый объект. Это означает, что модуль или диспетчер Web перенаправляет все поступающие сообщения, не требуя применять элементы Web-действия.

Компонент `WSDLHTMLPublish` помещают в Web-модуль приложения Web-службы, чтобы опубликовать список документов WSDL, описывающих приложение Web-службы. Публикуемые компонентом `WSDLHTMLPublish` определения выводятся из всех вызываемых интерфейсов, зарегистрированных с помощью реестра вызовов `InvRegistry`, как и из всех удаленных классов и типов, зарегистрированных в реестре удаленных типов `RemTypeRegistry`.

По умолчанию адрес, по которому установлено приложение, содержащее компонент `WSDLHTMLPublish`, используется как адрес, где эти службы доступны. В случае реализации Web-службы в другом месте, установив свойство `AdminEnabled` компонента в значение `True` и запустив WSDL-администратор из Web-обозревателя, можно изменить адреса, сгенерированные в документах WSDL.

Компонент `WSDLHTMLPublish` автоматически регистрирует себя с Web-модулем или Web-диспетчером как автодиспетчируемый объект. Это означает, что Web-модуль или Web-диспетчер пересылает все приходящие сообщения, не требуя применять элементы Web-действия. Эти сообщения включают запросы на список документов WSDL и запросы к WSDL-администратору.

Компонент `TTPSoapPascalInvoker` используют в приложении, которое публикует Web-службу с применением протокола SOAP. Этот компонент получает поступающий запрос SOAP от компонента `HTTPSoapDispatcher`, анализирует его, генерирует соответствующее обращение к зарегистрированному вызываемому интерфейсу и кодирует результаты вызова этого интерфейса.

В среде Delphi `HTTPSoapPascalInvoker` используют непосредственно как вызывающий компонент в приложении Web-службы. Этот компонент реализует интерфейс `IHTTPSoapDispatch`, который диспетчер использует для передачи входящих к нему запросов SOAP и приема содержимого ответного сообщения. Чтобы обеспечить диспетчеру возможность использовать этот интерфейс, нужно задать для свойства `Dispatcher` компонента `HTTPSoapPascalInvoker` значение `THTTPSoapDispatcher`.

Схема взаимодействия клиента и сервера

Схематично процесс взаимодействия клиента и сервера Web-службы приведен на рис. 35.2. Как видно из приведенной схемы, для того чтобы обратиться к серверу Web-службы, клиент с помощью программы импорта WSDL-документа предварительно должен получить описание требуемой службы в виде pas-файла.

На основе названного описания клиент Web-службы создает объект, который является своего рода посредником между клиентом и сервером при обращении именно к той службе, для которой он был создан. Функции объекта указаны на рис. 35.2.

Приведенная схема процесса взаимодействия между клиентом и сервером Web-службы является весьма упрощенной, поэтому на ней, в частности, не получили отображения различные способы передачи параметров.

Разработка клиента для Web-службы

В этой главе мы ограничимся рассмотрением вопроса создания клиента Web-службы. Во-первых, нам может и не потребоваться заниматься созданием собственного сервера Web-службы, а будет достаточно обеспечивать доступ к уже имеющимся службам. Во-вторых, создать приложение-клиент проще, нежели сервер. В-третьих, создание и использование клиента позволяет прояснить смысл программирования и использования Web-служб.

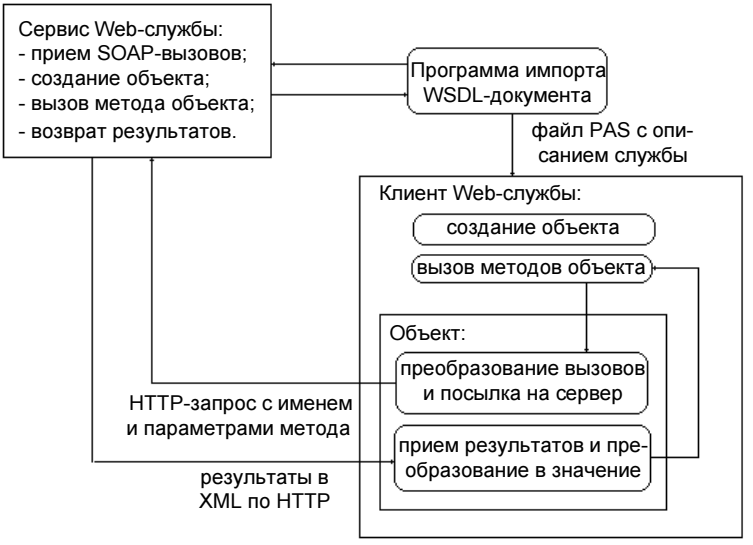


Рис. 35.2. Схема процесса взаимодействия клиента и сервера

Для разработки приложения-клиента, осуществляющего доступ к определенной в WSDL-документе Web-службе, требуется выполнить следующие шаги:

1. Импортирование определений из документа WSDL.
2. Получение вызываемого интерфейса и вызов его для доступа к Web-службе.
3. Может потребоваться обработка заголовков сообщений SOAP, передаваемых между клиентом и сервером.

Рассмотрим указанные шаги более подробно.

Импортирование документа WSDL

Для вызова программы импортирования документа WSDL нужно выбрать команду **File | New | Other** и на странице **WebServices** Хранилища объектов сделать двойной щелчок на значке **WSDL Importer**. В открывшемся диалоговом окне Мастера импорта WSDL, если известен адрес размещения документа WSDL в Интернете, следует указать его URL в поле **Location of WSDL file or URL**, нажать кнопку **Next** и после установления связи с провайдером импортировать его из Интернета. В противном случае с помощью кнопки **Search UDDI** откроем диалоговое окно **UDDI Browser** (рис. 35.3).

В раскрывающемся списке **Registry** нужно выбрать имя UDDI-реестра, который планируется использовать. При этом можно выбрать из списка общий (public) реестр или указать URL частного реестра.

В поле **Business Name** следует указать наименование отрасли бизнеса, которой принадлежит импортируемая Web-служба, например **software**.

В списке **Sort** нужно выбрать вариант сортировки результатов импорта документов.

Далее, нажав кнопку **Find**, инициируем поиск документа WSDL в Интернете с заданными параметрами. Дерево результатов поиска отображается в правой части диалогового окна **UDDI Browser** на вкладке **Results** (см. рис. 35.3). Здесь требуется выбрать подходящий вариант точки входа и после активизации кнопки **Import WSDL** сделать щелчок мышью. В результате должна происходить загрузка документа WSDL, описывающего выбранную Web-службу. В действительности вместо успешной загрузки порой выдается сообщение об ошибке (документ пуст).

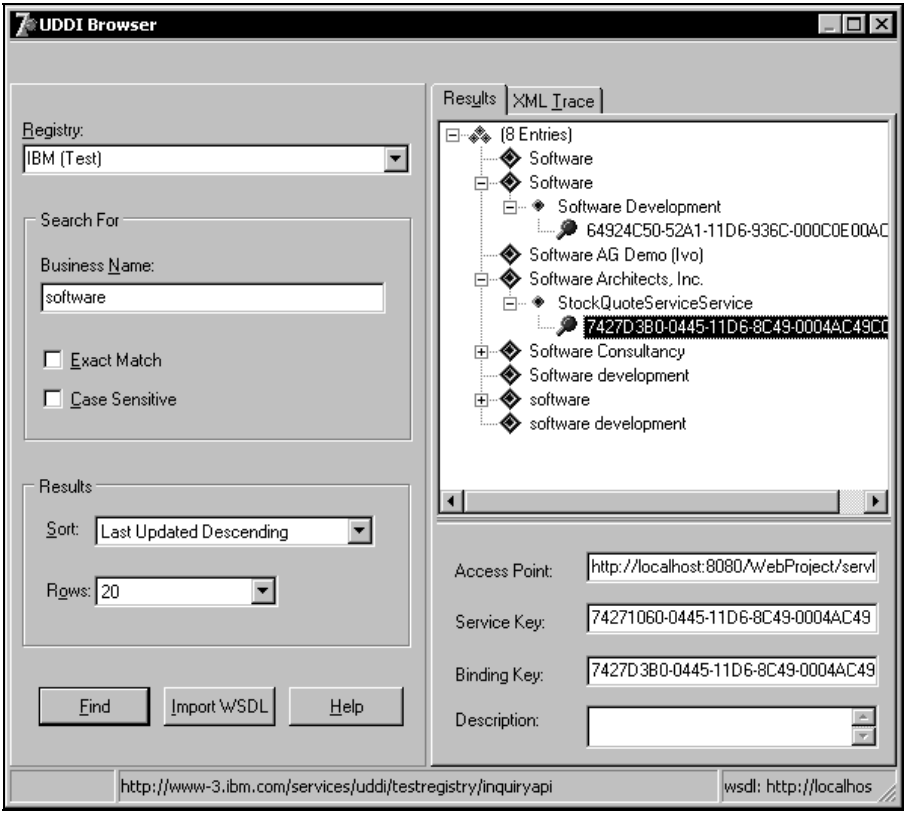


Рис. 35.3. Окно **UDDI Browser**

В случае успешной загрузки документа WSDL Мастер импорта выполняет его разбор и генерирует соответствующий модуль с кодом описания службы, помещая его в файл с расширением pas. Вид кода в правой части окна Мастера импорта приведен на рис. 35.4. В левой части этого окна расположено дерево типов и интерфейсов, имеющихся в документе WSDL. Работа с Мастером импорта документов WSDL завершается нажатием кнопки **Finish**.

Замечание

UDDI (Universal Description, Discovery and Integration, универсальное описание, обнаружение и объединение) представляет собой общий формат, используемый для регистрации Web-служб в Интернете.

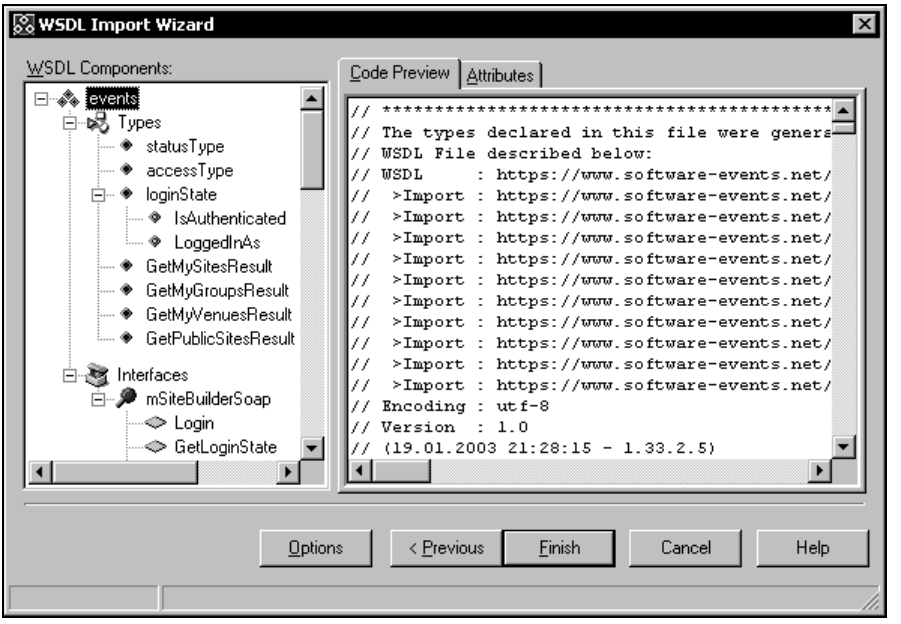


Рис. 35.4. Окно **WSDL Import Wizard** с компонентами документа WSDL

Обращение к вызываемому интерфейсу

Для *обращения к вызываемому интерфейсу* клиентское приложение должно включать все определения вызываемых интерфейсов и все удаляемые классы, реализующие сложные типы. Если сервер приложения Web-службы написан на Delphi, то можно использовать те же модули, которые сервер приложения использовал для определения и регистрации соответствующих интерфейсов и классов вместо файлов, генерируемых при импортировании документа WSDL. При этом нужно, чтобы модуль использовал то же самое пространство имен URI и заголовок SOAP Action при регистрации вызываемых интерфейсов. Эти значения можно или просто специфицировать в коде регистрации интерфейсов, или этот код может быть сгенерирован автоматически. Во втором случае определяющий интерфейс модуль должен иметь одно и то же имя у клиента и сервера, кроме того, клиент и сервер должны задать глобальной переменной `AppNameSpacePrefix` одинаковое значение.

При наличии определения вызываемого интерфейса экземпляр для вызова можно получить двумя описанными ниже способами.

При импортировании документа WSDL программа импортирования автоматически генерирует глобальную функцию, которая возвращает интерфейс и которую можно вызвать. Например, при импортировании документа WSDL, определяющего вызываемый интерфейс с именем `mSiteBuilderSoap` с помощью строки

```
mSiteBuilderSoap = interface(IInvokable)
```

сгенерированный модуль будет включать глобальную функцию с прототипом вида:

```
function GetmSiteBuilderSoap(UseWSDL: Boolean; Addr: string;
      HTTPRIO: THTTPRIO): mSiteBuilderSoap;
```

Если параметр `UseWSDL` имеет значение `True`, то местоположение сервера определяется с помощью документа WSDL, в противном случае клиентское приложение предоставляет URL Web-службы сервера с помощью параметра `Addr`. Если `UseWSDL` имеет значение `True`, то параметр `Addr` указывает URL документа WSDL, описывающего Web-службу.

При задании пустой строки для второго параметра названной функции по умолчанию будет использоваться импортированный документ. Такой подход предпочтителен в случаях возможного изменения URL для Web-службы или отдельных составляющих. Здесь информация просматривается динамически при вызове метода приложением.

Сгенерированная функция использует внутренний удаленный интерфейсный объект для реализации вызываемого интерфейса. Если применяется эта функция и выявлена необходимость доступа к лежащему в ее основе удаленному интерфейсному объекту, то интерфейс типа `IRIOAccess` можно получить из вызываемого интерфейса и использовать его для доступа к удаленному интерфейсному объекту:

```
var
  Interf: IServerInterface;
  RIOAccess: IRIOAccess;
  InterfObj: THTTPRIO;
begin
  Interf := GetIServerInterface(True,
         'http://MyServices.org/scripts/AppServer.dll/wsdl');
  RIOAccess := Interf as IRIOAccess;
  InterfObj := RIOAccess.RIO as THTTPRIO;
```

В случае использования удаленного интерфейсного объекта можно создать экземпляр (объект) типа `THTTPRio` для нужного интерфейса:

```
InterfObj := THTTPRio.Create(Nil);
```

Важно, что не требуется явно удалять интерфейсный объект типа `THTTPRio`. Если объект создан без владельца (Owner), как в последней строке кода, то он автоматически становится свободным при освобождении его интерфейса. Если он создан с помощью владельца, то владелец несет ответственность за освобождение экземпляра типа `THTTPRio`.

При наличии экземпляра типа `THTTPRio` следует снабдить его информацией, требуемой для идентификации интерфейса и размещения сервера. Это можно выполнить двумя путями.

- Если не предполагается, что URL, пространство имен или заголовок SOAP Action потребуется изменять, то можно просто задать URL для Web-службы, к которой нужен доступ. Объект типа `THTTPRio` использует этот URL для поиска определения интерфейса, а также информации о пространстве имен и заголовке на основе информации в реестре вызовов. URL задается путем установки свойства `URL` для размещения сервера:

  ```
  InterfObj.URL := 'http://www.myco.com/MyService.dll/SOAP/IServerInterface';
  ```

- Если предполагается динамическое определение URL, пространства имен или заголовка SOAP Action в процессе выполнения, то можно использовать свойства `WSDLLocation`, `Service` и `Port`. Это обеспечит извлечение необходимой информации из документа WSDL:

```
InterfObj.WSDLLocation := 'Cryptography.wsdl';
InterfObj.Service := 'Cryptography';
InterfObj.Port := 'SoapEncodeDecode';
```

После определения способа задания размещения сервера и идентификации интерфейса можно получить указатель интерфейса для вызываемого интерфейса из объекта типа `THTTPRio`. Для этого используется оператор `as`. Просто распределяется экземпляр объекта типа `THTTPRio` на вызываемый интерфейс:

```
InterfaceVariable := InterfObj as IEncodeDecode;
Code := InterfaceVariable.EncodeValue(5);
```

При получении указателя интерфейса объект типа `THTTPRio` создает `vtable` (массив указателей на методы — таблица виртуальных функций, ТВФ) для ассоциированного интерфейса динамически в памяти, поддерживая тем самым возможность вызовов интерфейса.

Объект типа `HTTPRio` для получения информации о вызываемом интерфейсе полагается на реестр вызовов. Если у клиентского приложения нет реестра вызовов или вызываемый интерфейс не зарегистрирован, то объект типа `HTTPRio` не может создать таблицу виртуальных функций (ТВФ) в памяти.

Если интерфейс, получаемый от объекта типа `HTTPRio`, назначен глобальной переменной, то перед завершением приложения нужно установить это назначение в `Nil`. К примеру, если `InterfaceVariable` из предыдущего примера есть глобальная переменная, а не переменная стека, то перед тем, как объект типа `HTTPRio` будет свободен, требуется освободить интерфейс. Обычно соответствующий код помещают в обработчик события `OnDestroy` формы или модуля данных:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  InterfaceVariable := Nil;
end;
```

Необходимость задания значения `Nil` для глобальной интерфейсной переменной обусловлена тем, что объект типа `HTTPRio` создает таблицу виртуальных функций в памяти динамически. Причем таблица виртуальных функций должна еще присутствовать, когда интерфейс освобожден. Если интерфейс не освобожден вместе с формой или с модулем данных, то он освобождается, когда освобождается глобальная переменная при завершении приложения. Память для глобальных переменных может быть освобождена после освобождения формы или модуля данных, содержащих объект типа `HTTPRio`, в этом случае таблица виртуальных функций не будет доступной, когда интерфейс свободен.

Пример создания клиента Web-службы

Рассмотрим пример создания клиента для Web-службы, размещенной по адресу **https://www.software-events.net/mSiteBuilderServices/events.asmx?WSDL**. При этом предположим, что предварительно мы создали проект приложения Delphi с пока еще пустой формой.

1. Выполним импортирование WSDL-документа: выберем команду **File | New | Other** и на странице **WebServices** Хранилища объектов сделаем двойной щелчок на значке

WSDL Importer. В открывшемся диалоговом окне Мастера импорта WSDL в поле **Location of WSDL file or URL** укажем адрес размещения документа WSDL в Интернете, нажмем кнопку **Next** и после установления связи с провайдером импортируем его.

Замечание

Напомним, что фрагменты документа WSDL для этой службы приведены в начале этой главы. Из текста документа видно, что вызываемый интерфейс имеет имя `mSiteBuilderSoap`, а автоматически сформированная глобальная функция — имя `GetmSiteBuilderSoap`. В описании интерфейса содержатся функции (`Login`, `GetLoginState`, `GetMySites` и др.), которые можно использовать при обращении к вызываемому интерфейсу Web-службы.

2. Для организации обращения создаваемого нами клиента к Web-службе поместим в форму кнопку и создадим для нее обработчик события `OnClick`, код которого приведен далее.

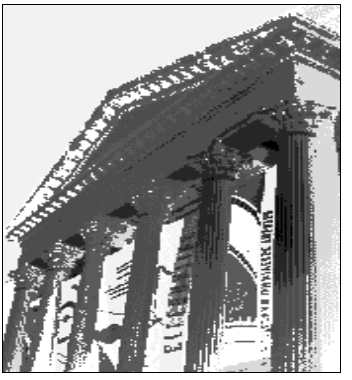
```
procedure TForm1.Button1Click(Sender: TObject);
var Port: mSiteBuilderSoap;
begin
  try
    Port := GetmSiteBuilderSoap;
    ShowMessage(BoolToStr(Port.Login('kkkk')));
  finally
    Port := Nil;
  end;
end;
end.
```

В приведенном коде переменной `Port` типа `mSiteBuilderSoap` присваивается значение глобальной функции `GetmSiteBuilderSoap`, выполняющей обращение к вызываемому интерфейсу. Далее с помощью функции `ShowMessage` отображается результат вызова функции `Login`, возвращающей значение типа `Boolean` и указывающей тем самым на возможность регистрации. При этом используется функция `BoolToStr`, преобразующая его в тип `String` (строка `'kkkk'` в качестве аргумента при обращении к функции взята произвольно).

Теперь при запуске созданного нами приложения-клиента после нажатия кнопки будет происходить открытие диалогового окна установления связи с провайдером, обращение к Web-службе и выдача сообщения о возможности регистрации (Login). При первом нажатии нашей кнопки появится окно сообщения (рис. 35.5) со значением `-1` (соответствует значению `True`), при всех последующих — со значением `0` (соответствует значению `False`).

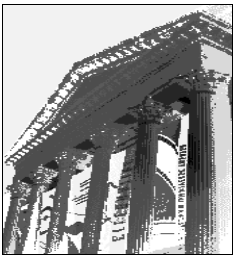
Описанным способом можно организовать обращение к любой функции в составе вызываемого интерфейса рассматриваемой Web-службы.

Рис. 35.5. Вид окна сообщения



ЧАСТЬ VII

Дополнительные возможности



ГЛАВА 36

Настройка параметров приложения

Многие приложения предоставляют пользователю возможность настраивать свои конфигурационные параметры, например, положение и размеры окон, коды доступа и пароли и т. д. Установленные значения параметров сохраняются на диске и могут быть считаны при последующем запуске приложения.

Информация, связанная с настройкой приложения и операционной системы Windows, обычно хранится в инициализационных файлах (расширение ini), а начиная с Windows 95 — также в системном реестре (Windows Registry).

Работа с инициализационными файлами

Инициализационный файл представляет собой текстовый файл, содержащий именованные разделы (секции). Имя раздела заключается в квадратные скобки. В каждом разделе находятся параметры, которым можно присваивать значения логического, целочисленного или строкового типов. Идентификатор параметра отделяется от значения знаком =. Выражение вида

```
<Идентификатор параметра> = <Значение>
```

называют ключевым или просто *ключом*.

Например, инициализационный файл schedule.ini содержит следующие разделы и строки (листинг 36.1).

Листинг 36.1. Инициализационный файл schedule.ini

```
[Scheduler]
ButtonStyle=838
StatusBar=1
Hide=0
Autoload=0
[NamedEvents]
```

```
VirusScan=1
LiveUpdate=1
[VirusScan]
Name=Поиск вирусов
Prompt=&Где искать:
CommandLine=NAVW32.EXE
DefaultDesc=Поиск вирусов
DefaultActionText=/L
AllowBlankActionText=0
BlankActionTextMessage=Введите допустимый путь для проверки или "/L"
 для поиска на всех локальных дисках.
StartupDir=
RunStyle=1
[LiveUpdate]
Name=Запуск LiveUpdate (или Norton AntiVirus)
Prompt=&Командная строка ("/Prompt" для запроса перед запуском):
CommandLine=NAVLU32.EXE /SCHEDULED
DefaultDesc=Запуск LiveUpdate (или Norton AntiVirus)
StartupDir=
RunStyle=0
```

Для работы с инициализационными файлами в Delphi предназначен класс `TIniFile`, являющийся прямым потомком класса `TObject`. При использовании в приложении класса `TIniFile` в разделе `uses` модуля, выполняющего операции с инициализационным файлом, следует подключить модуль `IniFiles`.

Работа с инициализационным файлом ведется как с обычным текстовым файлом: в программе объявляется файловая переменная (логический файл), которая затем связывается с физическим файлом на диске. После установления связи с помощью процедур `write` (`writeln`) и `read` (`readln`) выполняется запись данных на диск и чтение их с диска. После завершения операций ввода/вывода связь между файловой переменной и файлом на диске разрывается.

Для работы с инициализационным файлом описывается специальная переменная типа `TIniFile`. Перед действиями с инициализационным файлом следует создать объект типа `TIniFile`, для чего используется метод `Create`. Конструктор `Create(const FileName String)` создает в приложении экземпляр класса `TIniFile`, связывая его с инициализационным файлом, имя которого задано параметром `FileName`. Если указанный файл на диске отсутствует, он создается заново.

Если в имени файла полный путь не указан, то по умолчанию инициализационный файл ищется в системном каталоге Windows, который может иметь имя C:\Windows или C:\WinNT. С одной стороны, это удобно, т. к. многие инициализационные файлы хранятся в общем месте, с другой стороны, этот файл находится отдельно от других файлов приложения, которые обычно расположены в специальном каталоге. Поэтому при удалении приложения конфигурационный файл часто остается неудаленным и занимает место на диске. С этой точки зрения целесообразно располагать инициализационный файл в одном каталоге с остальными файлами приложения.

Для получения имени инициализационного файла, связанного с экземпляром класса `TIniFile`, предназначено свойство `FileName` типа `String`. Оно действует во время выполнения программы и доступно для чтения.

Для освобождения занимаемой экземпляром `TIniFile` памяти после завершения операций с инициализационным файлом используется метод `Free`.

Приведем пример использования объекта класса `TiniFile`.

```
// В разделе uses должен быть указан модуль IniFiles
var ConfigFile: TIniFile;
...
ConfigFile := TIniFile.Create('Config.ini');
// Местоположение инструкций, выполняющих
// операции с файлом Config.ini
ConfigFile.Free;
```

Здесь создается и освобождается переменная `ConfigFile`, которая связана с файлом Config.ini.

Между вызовами методов `Create` и `Free` располагаются инструкции, выполняющие операции с инициализационным файлом, например, чтение (`ReadXXX`)/запись (`WriteXXX`) заданного раздела значений параметров.

Для *чтения* параметров используются методы `ReadBool`, `ReadFloat`, `ReadInteger`, `ReadString`, `ReadDate`, `ReadTime` и `ReadDateTime`. Эти методы являются функциями и имеют описание следующего формата:

```
function ReadXXX(Раздел, Идентификатор, Значение по умолчанию): Тип результата
```

Например, заголовок метода `ReadBool` имеет такой вид:

```
function ReadBool(Const Section, Ident: String; Default: Boolean):
  Boolean; virtual;
```

Остальные методы `ReadXXX` отличаются только типами возвращаемого результата и параметра `Default`. Каждая функция возвращает значение параметра `Ident` из раздела `Section`. Параметр `Default` задает значение по умолчанию, возвращаемое как результат, если идентификатор не найден. Значения строкового (`String`), целочисленного (`Longint`), вещественного (`Double`) типов, типа даты и времени (`TDateTime`) кодируются как обычно, а для значений логического (`Boolean`) типа используются ноль (`False`) и единица (`True`).

Рассмотрим в качестве примера использование файла prog.ini для настройки конфигурационных параметров приложения. При выполнении приложения из файла читаются значения трех таких параметров. Параметр `CountStart` раздела `START` содержит число предыдущих запусков программы, которое при каждом запуске увеличивается на единицу и отображается в надписи `Label1`. В разделе `USER` параметр `UserName` содержит фамилию пользователя, заносимую в поле редактирования `Edit1`, а параметр `RussianLanguage` задает язык надписей на кнопках `Button1` и `Button2`. В заголовке формы выводится имя инициализационного файла (рис. 36.1).

В листинге 36.2 приведен исходный код модуля формы для работы с инициализационным файлом.

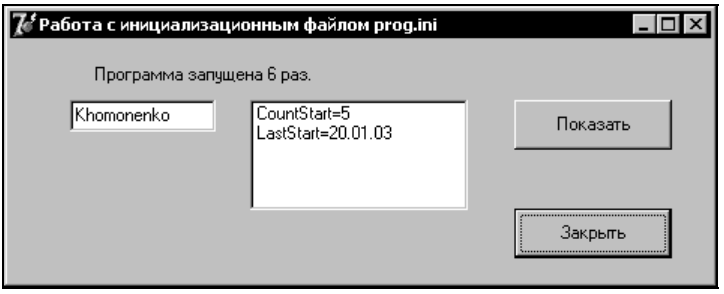


Рис. 36.1. Форма для работы с инициализационным файлом

Листинг 36.2. Пример работы с инициализационным файлом

```
// Подключение модуля для объекта типа TIniFile
uses IniFiles;
...
var n, cr: Longint;
       rl: Boolean;
. . .
procedure TForm1.FormCreate(Sender: TObject);
var cfProg  :TIniFile;
begin
  // Файл prog.ini ищется в системном каталоге Windows
  cfProg := TIniFile.Create('prog.ini');
  try
    try
      n := cfProg.ReadInteger('START', 'CountStart', 0);
      Label1.Caption := 'Программа запущена ' + IntToStr(n + 1) + ' раз.';
      Edit1.Text := cfProg.ReadString('USER', 'UserName', '');
      rl := cfProg.ReadBool('USER', 'RussianLanguage', True);
      if rl then begin
        Button1.Caption := 'Показать';
        Button2.Caption := 'Закрыть';
      end
      else begin
        Button1.Caption := 'Show';
        Button2.Caption := 'Close';
      end;
      cr := cfProg.ReadInteger('START', 'Code', 10);
      Form1.Caption := 'Работа с инициализационным файлом ' +
                       cfProg.FileName;
    finally
      cfProg.Free;
    end;
  except
    MessageDlg('Ошибка настройки конфигурации!', mtError, [mbYes], 0);
  end;
end;
```

При чтении данных из инициализационного файла обеспечивается обработка исключений.

Сам инициализационный файл prog.ini может иметь следующий вид:

```
[USER]
UserName=Khomonenko
RussianLanguage=1
[START]
CountStart=6
LastDate=20.01.03
```

В приведенном примере читается целочисленное значение параметра `Code`, который отсутствует в указанном разделе `START`. Поэтому переменной `cr` присваивается значение `10` как значение по умолчанию. В то же время раздел `START` содержит идентификатор `LastDate`, значение которого в приложении не используется.

Рассмотренные методы считывают значение одного параметра. Метод `ReadSectionValues(const Section: string; Strings: TStrings)` позволяет прочитать значения *всех параметров* заданного раздела инициализационного файла. При его выполнении все ключевые выражения из раздела `Section` считываются и загружаются в строковый объект `Strings`, где они запоминаются в таком же виде и порядке, как в инициализационном файле.

Например, в процедуре

```
procedure TForm1.Button2Click(Sender: TObject);
var cfProg: TIniFile;
begin
  cfProg := TIniFile.Create('prog.ini');
  cfProg.ReadSectionValues('START', ListBox1.Items);
  cfProg.Free;
end;
```

при нажатии кнопки `Button2` с заголовком **Показать** содержимое раздела `START` инициализационного файла prog.ini выводится в списке `ListBox1`. Отметим, что здесь не предусмотрена локальная обработка исключений, поэтому ошибки, возможные при операциях с инициализационным файлом, обрабатываются глобальным обработчиком приложения.

После считывания содержимого раздела его отдельные элементы, находящиеся в списке, можно обрабатывать обычными для списков способами.

Для *записи* значений различных типов в инициализационный файл предназначены методы `WriteBool`, `WriteFloat`, `WriteInteger`, `WriteString`, `WriteDate`, `WriteTime` и `WriteDateTime`. Эти методы являются процедурами и имеют похожее действие и формат:

```
function WriteXXX(Раздел, Идентификатор, Значение)
```

Например, процедура `WriteBool(const Section, Ident: string; Value: Boolean)` записывает для параметра `Ident` логическое значение `Value`. Параметр `Section` указывает раздел файла. Остальные процедуры `WriteXXX` отличаются только типом параметра `Value`.

Если указанные в процедуре WriteXXX раздел или идентификатор отсутствуют, то они автоматически создаются.

Например, в процедуре закрытия формы:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var cfProg : TIniFile;
begin
  cfProg := TIniFile.Create('prog.ini');
  cfProg.WriteString('USER', 'UserName', Edit1.Text);
  cfProg.WriteInteger('START', 'CountStart', n + 1);
  cfProg.WriteBool('USER', 'RussianLanguage', rl);
  cfProg.WriteString('START', 'LastDate', DateToStr(Date));
  cfProg.Free;
end;
```

в соответствующих разделах инициализационного файла prog.ini сохраняются значения четырех параметров: фамилия пользователя UserName, число запусков приложения CountStart, признак RussianLanguage, задающий русский язык для надписей, отображаемых на кнопках, и дата LastDate последнего запуска приложения. В связи с тем, что в инициализационный файл нельзя записать значение типа "дата", текущая дата преобразуется в строковый тип.

Отдельные составляющие инициализационного файла можно *удалять* с помощью методов EraseSection и DeleteKey, при этом процедура EraseSection(const Section: string) удаляет из файла целый раздел, указанный параметром Section, а процедура DeleteKey(const Section, Ident: string) удаляет из раздела Section ключевое выражение, соответствующее параметру Ident, включая идентификатор параметра. Если удаляемый элемент не найден, то обе процедуры никакого действия не выполняют. Например, при выполнении приведенной далее процедуры:

```
procedure TForm1.Button3Click(Sender: TObject);
var cfProg: TIniFile;
begin
  cfProg := TIniFile.Create(ExtractFilePath(Application.ExeName) +
          'test.ini');
  cfProg.EraseSection('NAME');
  cfProg.DeleteKey('DATA', 'ID2');
  cfProg.Free;
end;
```

из инициализационного файла test.ini удаляются раздел NAME и идентификатор ID2 раздела DATA. Инициализационный файл расположен в каталоге приложения.

Класс TIniFile не имеет специальных методов, предназначенных для добавления в инициализационный файл разделов и ключевых выражений. Напомним, что эти действия автоматически выполняют методы WriteXXX, которые создают требуемый раздел и/или записывают ключевое выражение в случае их отсутствия.

Получить список имен разделов или имен параметров некоторого раздела позволяют методы ReadSections и ReadSection. Как и процедура ReadSectionValues, они считывают список имен разделов инициализационного файла и список идентификаторов

указанного раздела соответственно, а затем загружают эти имена в строковый объект типа `TStrings`, например, в список `ListBox`.

Работа с системным реестром

Системный реестр Windows представляет собой специальную базу данных, предназначенную для централизованного хранения разнообразной информации об операционной системе и приложениях.

Принципы работы и организации данных в системном реестре и инициализационном файле Windows во многом аналогичны. Оба содержат значения параметров, которые можно различными способами записывать или считывать. Отличительной особенностью системного реестра является расположение информации в иерархическом виде с несколькими основными (корневыми) разделами ("ульями" — hives). Для работы с системным реестром в Windows предназначена программа Редактор реестра (Registry Editor). Вид основных разделов системного реестра в окне Редактора реестра показан на рис. 36.2.

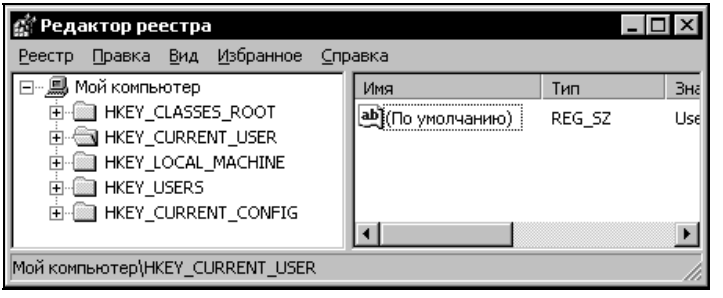


Рис. 36.2. Основные разделы системного реестра

Основные разделы содержат различные группы (ключи — keys) системной информации, которые, в свою очередь, могут также содержать подразделы (подключи — subkeys) и параметры.

Замечание

В отличие от системного реестра Windows, в инициализационных файлах под "ключом" понимается выражение вида "идентификатор = значение". Эти понятия нельзя путать.

Параметр реестра имеет имя и значение допустимого типа: строкового, логического, целочисленного или вещественного. Структура ключей и параметров реестра напоминает файловую структуру диска (каталоги и файлы). По соглашению имена основных разделов начинаются с префикса `HKEY` (от HiveKEY). Каждый основной раздел содержит определенную группу данных. Например, в разделе `HKEY_LOCAL_MACHINE` хранится информация об аппаратном и программном обеспечении локального компьютера.

Иерархическое расположение конфигурационной информации создает некоторые трудности при ее поиске, но позволяет избавиться от большого количества отдельных инициализационных файлов.

Замечание

Существует мнение, что в системах Windows 95/98/2000/NT информацию о настройке приложения нужно сохранять не в инициализационном файле, что является устаревшим подходом, а именно в системном реестре Windows. При использовании системного реестра следует иметь в виду, что он содержит и информацию о параметрах других приложений, включая саму систему Windows, а также информацию о параметрах компьютера. В связи с тем, что нарушение системного реестра может разрушить систему, операции с ним необходимо выполнять с большой осторожностью и вниманием.

Для работы с системным реестром в Delphi предназначен класс `TRegistry`. Чтобы получить доступ к свойствам и методам этого класса, в разделе `uses` программы следует указать модуль `Registry`.

Экземпляр класса `TRegistry` создается и удаляется с помощью методов `Create` и `Free`, между которыми располагаются инструкции, выполняющие требуемые операции с разделами и ключами реестра. В отличие от создания экземпляра объекта для инициализационного файла, конструктор `Create` не имеет параметра, т. к. переменная типа `TRegistry` всегда ассоциируется с системным реестром. Например:

```
var regSaleRead: TRegistry;
...
regSaleRead:=TRegistry.Create;
// В этом месте располагаются инструкции обработки реестра
regSaleRead.Free;
```

Для указания *текущего корневого раздела* ("улья") используется свойство `RootKey` типа `HKEY`. По умолчанию свойство `RootKey` имеет значение `HKEY_CURRENT_USER`; другой ключ при необходимости можно задать, например, так:

```
regTest.RootKey := HKEY_LOCAL_MACHINE;
```

Тип свойства `RootKey` не является строковым, поэтому его значение в апострофы не заключается.

Для определения *текущего ключа* служат свойства `CurrentKey` типа `HKEY` и `CurrentPath` типа `String`, которые доступны для чтения при выполнении программы. Свойство `CurrentKey` содержит имя текущего ключа, а свойство `CurrentPath` — полный путь к нему.

Все операции выполняются с текущим ключом, который является открытым. Для выполнения какой-либо операции с другим ключом следует перейти к этому ключу, т. е. *открыть* ключ. Для этого служит метод `OpenKey`. Функция `OpenKey(const Key: String; CanCreate: Boolean): Boolean` пытается открыть ключ, заданный параметром `Key`. Если этот ключ существует, то он становится текущим, в противном случае учитывается параметр `CanCreate`. При значении `True` этого параметра ключ создается и становится текущим. Если указанный параметром `Key` ключ открыт, то функция `OpenKey` возвращает значение `True`, в противном случае — `False`.

После завершения работы с ключом следует вызвать метод `CloseKey`, который *закрывает* текущий ключ, а его значение записывает в системный реестр. Если открытого раздела нет, то метод `CloseKey` игнорируется.

С учетом обработки возможных исключений для доступа к системному реестру обычно используется следующая последовательность инструкций:

```
var rgExample: TRegistry;
...
rgExample := TRegistry.Create;
rgExample.RootKey := HKEY_LOCAL_MACHINE;
try
  if rgExample.OpenKey('\SOFTWARE\BookStore', False) then begin
    // Инструкции обработки реестра
    rgExample.CloseKey;
  end;
finally
  rgExample.Free;
end;
```

Для *чтения* из системного реестра информации различного типа используются методы `ReadXXX`: `ReadBool`, `ReadCurrency`, `ReadDate`, `ReadDateTime`, `ReadFloat`, `ReadInteger`, `ReadString`, `ReadTime`, `ReadBinaryData`.

Перечисленные функции по описанию и действию подобны функциям, используемым для чтения информации из инициализационного файла. Например, функция `ReadBool(const Name: String): Boolean` читает из текущего (открытого) ключа и возвращает как результат значение логического типа, имя которого задано параметром `Name`. Другие функции `ReadXXX`, кроме `ReadBinaryData`, отличаются от `ReadBool` только типом возвращаемого результата, указанного в названии метода, и предназначены для чтения из реестра данных "своих" типов. Что касается функции `ReadBinaryData(const Name: String; var Buffer; BufSize: Integer): Integer`, то она считывает из инициализационного файла в буфер, заданный параметром `Buffer`, блок двоичных данных, длину которого определяет параметр `BufSize`.

В методах чтения данных из реестра отсутствует параметр, присваивающий значение по умолчанию. Если заданное параметром `Name` имя в указанном разделе (ключе) не существует, то возникает исключение.

Пример чтения данных из системного реестра приведен в листинге 36.3.

Листинг 36.3. Пример чтения данных из системного реестра

```
procedure TForm1.Button1Click(Sender: TObject);
var regTest2 :TRegistry;
begin
  regTest2 := TRegistry.Create;
  regTest2.RootKey := HKEY_LOCAL_MACHINE;
  try
    if regTest2.OpenKey('\SOFTWARE\Test', False) then begin
      Form1.Caption := regTest2.ReadString('FormCaption');
      regTest2.CloseKey;
    end;
  finally
    regTest2.Free;
  end;
end;
```

Здесь выполняется чтение параметра строкового типа с именем `FormCaption` из ключа `SOFTWARE\Test`. Считанное значение выводится в качестве заголовка формы.

Если необходимо *определить тип* данных, содержащихся в некотором ключе, то можно использовать методы `GetDataType` или `GetDataInfo`.

Для *записи нового значения* параметра в реестр служат методы `WriteXXX`: `WriteString`, `WriteInteger`, `WriteBinaryData`, `WriteCurrency`, `WriteDate`, `WriteFloat`, `WriteBool`, `WriteTime`.

Перечисленные функции по описанию и действию подобны функциям, используемым для записи информации в инициализационный файл. Так, процедура `WriteBool(const Name: String; Value: Boolean)` работает с текущим (открытым) ключом и для параметра с именем `Name` записывает логическое значение `Value`. Другие процедуры `WriteXXX`, кроме `WriteBinaryData`, отличаются от `WriteBool` только типом запоминаемого значения. Функция `WriteBinaryData(const Name: String; var Buffer; BufSize: Integer)` записывает в инициализационный файл из буфера, заданного параметром `Buffer`, блок двоичных данных.

Если параметр с указанным именем в текущем разделе отсутствует, то он создается автоматически. После записи данных ключ следует закрыть для обновления информации в реестре.

Пример записи данных в системный реестр приведен в листинге 36.4.

Листинг 36.4. Пример записи данных в системный реестр

```
procedure TForm1.Button2Click(Sender: TObject);
var regTest2  :TRegistry;
begin
  regTest2 := TRegistry.Create;
  regTest2.RootKey := HKEY_LOCAL_MACHINE;
  try
    if regTest2.OpenKey('\SOFTWARE\Test', False) then begin
      regTest2.WriteBool('VisibleButton', True);
      regTest2.WriteInteger('Code', 137);
      regTest2.CloseKey;
    end;
  finally
    regTest2.Free;
  end;
end;
```

Здесь выполняется запись логического и целочисленного значений в параметры реестра `VisibleButton` и `Code`, находящиеся в ключе `SOFTWARE\Test`.

Для *создания ключа* используется функция `CreateKey(const Key: String): Boolean`. В случае успешного создания ключа, заданного параметром `Key`, возвращается значение `True`, в противном случае — `False`. Если ключ с таким именем существует, то метод `CreateKey` не действует.

Определить, *существует ли ключ*, можно с помощью метода `KeyExists(const Key: String): Boolean`, который проверяет наличие в реестре ключа с именем `Key`. Если ключ найден, возвращается значение `True`, в противном случае — `False`.

В качестве примера рассмотрим процедуру, проверяющую существование системного ключа:

```
procedure TForm1.Button3Click(Sender: TObject);
var regTest3  :TRegistry;
begin
  regTest3 := TRegistry.Create;
  regTest3.RootKey := HKEY_LOCAL_MACHINE;
  try
    if regTest3.OpenKey('\SOFTWARE\Лупа-МБ\', True) then
      if not regTest3.KeyExists('New') then
          regTest3.CreateKey('New');
  finally
    regTest3.Free;
  end;
end;
```

Ключи создаются двумя способами. Если ключ `Лупа-МБ` отсутствует, то он создается при попытке сделать его текущим, что выполняет метод `OpenKey` (второй параметр — `CanCreate` — этого метода имеет значение `True`). Ключ `New` предварительно проверяется на существование с помощью вызова метода `KeyExists`, и в случае отсутствия создается методом `CreateKey`.

Для *удаления ключа* служит метод `DeleteKey`. Функция `DeleteKey(const Key: string): Boolean` удаляет ключ, указанный параметром `Key`, и все содержащиеся в нем подключи. Если удаление прошло успешно, то в качестве результата возвращается значение `True`, в противном случае — `False`.

Кроме уже рассмотренных, у класса `TRegistry` есть много других методов, позволяющих выполнять с объектами этого класса различные операции, например, переименовать данные или получить информацию о текущем ключе.

Рассмотрим пример того, как можно использовать реестр для конфигурирования приложения.

Код модуля `uRegistry` главной формы приложения имеет следующий вид (листинг 36.5).

Листинг 36.5. Использование реестра для конфигурирования приложения

```
unit uRegistry;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Label1  : TLabel;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;
var
  Form1  : TForm1;
implementation
// Подключение модуля для объекта TRegistry
uses Registry;
{$R *.DFM}

// Запоминание в реестре конфигурационной информации
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var regSaleWrite: TRegistry;
begin
  regSaleWrite := TRegistry.Create;
  regSaleWrite.RootKey := HKEY_LOCAL_MACHINE;
  try
    if regSaleWrite.OpenKey('\SOFTWARE\Лура-МБ\', True) then begin
      if not regSaleWrite.KeyExists('Sale') then
         regSaleWrite.CreateKey('Sale');
      if regSaleWrite.OpenKey('\SOFTWARE\Лура-МБ\Sale', False) then
        begin
          regSaleWrite.WriteInteger('MainFormWidth', Form1.Width);
          regSaleWrite.WriteInteger('MainFormHeight', Form1.Height);
          regSaleWrite.WriteInteger('MainFormTop', Form1.Top);
          regSaleWrite.WriteInteger('MainFormLeft', Form1.Left);
          regSaleWrite.WriteDateTime('LastDateTime', Now);
          regSaleWrite.CloseKey;
        end;
    end;
  finally
    regSaleWrite.Free;
  end;
end;

// Загрузка и использование конфигурационной информации из реестра
procedure TForm1.FormCreate(Sender: TObject);
var regSaleRead: TRegistry;
begin
  regSaleRead := TRegistry.Create;
  regSaleRead.RootKey := HKEY_LOCAL_MACHINE;
  try
    if regSaleRead.OpenKey('\SOFTWARE\Лура-МБ\Sale', False) then begin
       Form1.Width := regSaleRead.ReadInteger('MainFormWidth');
       Form1.Height := regSaleRead.ReadInteger('MainFormHeight');
       Form1.Top := regSaleRead.ReadInteger('MainFormTop');
       Form1.Left := regSaleRead.ReadInteger('MainFormLeft');
       Label1.Caption := Дата и время окончания предыдущего сеанса работы' +
             DateTimeToStr(regSaleRead.ReadDateTime('LastDateTime'));
       regSaleRead.CloseKey;
     end;
```

```
  finally
    regSaleRead.Free;
  end;
end;
end.
```

Здесь в системном реестре запоминается информация о расположении и размерах главного окна, а также дата и время окончания сеанса работы с приложением. Каждая из двух процедур `FormClose` и `FormCreate` использует свою переменную типа `TRegistry` (вместо этого можно было объявить и использовать одну глобальную переменную этого же типа).

Рассмотренные операции с системным реестром Windows можно применять при создании приложений для Windows 95/98/2000 и Windows NT. При этом следует учитывать, что построение реестров в этих операционных системах несколько отличается друг от друга, например, это касается именования отдельных ключей.

Пример настройки параметров приложения

В качестве примера рассмотрим приложение для просмотра и редактирования таблиц БД (рис. 36.3). Пользователь может выбирать таблицу, данные которой выводятся в компоненте `DBGrid1`. Выбор осуществляется в диалоге открытия файла, вызываемом нажатием кнопки **Таблица**. После выбора главного файла таблицы его имя устанавливается в качестве значения свойства `TableName` набора данных `Table1`. Диалог открытия файла представлен компонентом `OpenDialog1`.

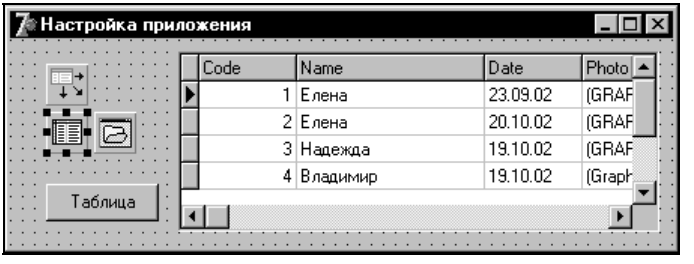


Рис. 36.3. Форма приложения

Приложение при закрытии сохраняет в своем инициализационном файле Config.ini имя файла таблицы, с которой выполнялись операции. Инициализационный файл находится в одном каталоге с приложением. Имя таблицы запоминается с помощью обработчика события закрытия главной формы. При последующем запуске приложения набор данных связывается с таблицей, которая была открыта последний раз. Имя таблицы считывается из инициализационного файла. Это действие выполняется в обработчике события создания формы. Здесь же устанавливается фильтр (свойство `Filter`) для диалога `OpenDialog1`.

В листинге 36.6 приведен код модуля `Unit1` главной формы `Form1` приложения.

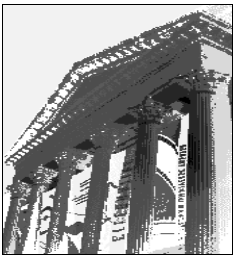
Листинг 36.6. Пример настройки параметров приложения

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Db, DBTables, Grids, DBGrids;
type
TForm1 = class(TForm)
     DBGrid1: TDBGrid;
 DataSource1: TDataSource;
      Table1: TTable;
     Button1: TButton;
 OpenDialog1: TOpenDialog;
 procedure FormCreate(Sender: TObject);
 procedure Button1Click(Sender: TObject);
 procedure FormClose(Sender: TObject; var Action: TCloseAction);
private
 { Private declarations }
public
 { Public declarations }
end;
var Form1: TForm1;
 ApplPath: String;
implementation
// Подключение модуля для работы с инициализационным файлом
uses IniFiles;
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var ConfigFile: TIniFile;
          name: String;
begin
  // Задание фильтра для выбора файлов
  OpenDialog1.Filter:='Таблицы(*.db;*.dbf)|*.db;*.dbf|Все файлы|*.*';
  // Определение и запоминание каталога приложения
  ApplPath := ExtractFilePath(Application.ExeName);
  // Чтение имени таблицы из инициализационного файла
  ConfigFile := TIniFile.Create(ApplPath + 'Config.ini');
  name := ConfigFile.ReadString('TABLE', 'WorkTable', '');
  ConfigFile.Free;
  // Открытие таблицы
  Table1.Active     := False;
  Table1.TableName  := name;
  Table1.Active     := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    Table1.Active    := False;
```

```
    Table1.TableName := OpenDialog1.FileName;
    Table1.Active    := True;
  end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var ConfigFile: TIniFile;
begin
  // Сохранение имени таблицы в инициализационном файле
  ConfigFile := TIniFile.Create(ApplPath + 'Config.ini');
  ConfigFile.WriteString('TABLE', 'WorkTable', Table1.TableName);
  ConfigFile.Free;
end;
end.
```

Аналогичным способом можно сохранять и другие настройки приложения БД, например, состав и цвета столбцов компонента `DBGrid`, поля, включенные в состав компонента-отчета `QReport`, список выбора для поля и т. д.

ГЛАВА 37



Организация обмена данными

Часто для обеспечения взаимодействия различных приложений или частей одного приложения организуется обмен данными. Операционная система Windows предоставляет для этого следующие технологии и средства:

- использование буфера обмена;
- динамический обмен данными;
- связывание и внедрение объектов, или технология OLE (Object Linking and Embedding).

В этой главе мы рассмотрим первые два подхода как достаточно распространенные и относительно просто реализуемые.

Работа с буфером обмена

Буфер обмена (Clipboard), или просто буфер представляет собой область оперативной памяти, которая используется операционной системой Windows для временного хранения данных. Буфер обмена является общим для всех программ, любое приложение может помещать в него информацию и считывать ее оттуда. Буфер обмена способен хранить данные самых разных типов и, кроме собственно данных, содержит сведения об их формате. Буфер обмена реализует *статический* обмен данными.

Для выполнения *операций обмена* данными через буфер в Delphi предназначен специальный класс `TClipboard`. В Delphi также имеется *глобальный объект* `Clipboard`, являющийся экземпляром класса `TClipboard` и представляющий собой буфер обмена Windows.

С помощью свойств и методов объекта `Clipboard` при работе с буфером обмена можно выполнять стандартные операции, например, очистку буфера или определение типа хранимых данных. Для доступа к объекту буфера обмена в разделе `uses` модуля, в котором выполняются операции с объектом буфера, указывается модуль `Clipbrd`.

Буфер обмена поддерживает различные форматы данных, общее число которых содержит свойство `FormatCount` типа `Integer`. Это свойство доступно для чтения при выполнении программы.

Пример отображения числа форматов буфера обмена:

```
uses Clipbrd;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(Clipboard.FormatCount);
end;
```

Список форматов, поддерживаемых буфером обмена, содержит свойство `Formats[Index: Integer]` типа `Word`. Свойство `Formats` является массивом и доступно для чтения во время выполнения программы. Форматы в списке представляются целыми значениями. Некоторые из возможных значений свойства `Formats` и соответствующие им именованные константы приведены в табл. 37.1.

Таблица 37.1. *Значения свойства* `Formats`

Значение	Константа	Формат буфера обмена
1	`CF_Text`	Обычный текст (коды ANSI). Каждая строка заканчивается символами с кодами `#10` и `#13`, в конце текста находится символ `#0`
2	`CF_Bitmap`	Рисунок формата BMP
3	`CF_MetaFilePict`	Рисунок формата WMF
6	`CF_TIFF`	Рисунок формата TIFF
7	`CF_OEMText`	Текст
13	`CF_UniCodeText`	Текст (коды Unicode)
53 457	`CF_Picture`	Объект типа `TPicture`

Первые пятнадцать значений принадлежат стандартным форматам, остальные обозначают дополнительные форматы, одним из которых является формат `CF_Picture`. При необходимости можно создать и зарегистрировать свои форматы данных в дополнение к имеющимся базовым.

Рассмотрим пример. В процедуре

```
procedure TForm1.Button2Click(Sender: TObject);
var n: Integer;
begin
  ListBox1.Clear;
  for n := 0 to Clipboard.FormatCount - 1 do
    ListBox1.Items.Add(IntToStr(Clipboard.Formats[n]));
end;
```

значения форматов буфера обмена загружаются в список `ListBox1`.

Приложение может помещать информацию в буфер обмена и извлекать ее только в тех форматах, которые буфер поддерживает. Список поддерживаемых форматов создается при инициализации приложения.

При использовании нестандартных форматов данных, помещаемых в буфер обмена и извлекаемых из него, программы должны соблюдать устанавливаемые разработчиками соглашения об обмене данными.

Перед доступом к данным, содержащимся в объекте `Clipboard`, может потребоваться *информация* о *формате данных*, для получения которой используется метод `HasFormat`. Функция `HasFormat(Format: Word): Boolean` служит для запроса к буферу обмена, можно ли извлечь хранимые в нем данные в формате, указанном параметром `Format`. При положительном ответе функция возвращает значение `True` и `False` — в противном случае.

Буфер обмена часто используется для хранения текста, поэтому объект `Clipboard` имеет специальное свойство `AsText` типа `String`, предназначенное для *обработки* содержимого буфера как *данных текстового формата*. Свойство `AsText` предназначено для *чтения*, что соответствует извлечению текста из буфера обмена, и для *записи* текста в буфер.

Пример копирования текстовых данных из буфера:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  if Clipboard.HasFormat(CF_Text)
    then Memo1.Text := Clipboard.AsText
    else MessageDlg('В буфере обмена — не текст!', mtError, [mbOK], 0);
end;
```

Если в буфере содержатся текстовые данные, то они с помощью метода `AsText` копируются в компонент `Memo1`, в противном случае выдается сообщение об ошибке.

Для *очистки* содержимого буфера обмена используется метод `Clear`. Этот метод вызывается автоматически при изменении содержимого буфера, удаляя предыдущие данные перед тем, как на их место записать новые. Метод `Clear` можно использовать для очистки буфера программным способом, например, в случае, когда пользователь приложения поместил в буфер обмена большой фрагмент данных и, закончив работу с приложением, не удалил его.

Обычно буфер обмена используется разными приложениями на равных правах. В случае, когда приложение хочет получить монопольный доступ к буферу обмена, оно должно открыть его для себя в соответствующем режиме. Для этого вызывается метод `Open`, позволяющий приложению получить *исключительный доступ* к общей области обмена. После вызова метода `Open` содержимое буфера не может быть изменено другими приложениями, поэтому по окончании монопольного использования буфера приложение должно вызвать для объекта `Clipboard` метод `Close`. Метод `Close` закрывает монопольный доступ приложения к буферу обмена и позволяет получить доступ к нему другим приложениям. Если открытый буфер не был закрыт с помощью метода `Close`, то он будет автоматически закрыт системой после завершения приложения, открывшего буфер обмена.

Класс `TClipboard` используется многими другими классами и компонентами, которые поддерживают обмен данными через буфер обмена.

Например, у компонентов `Memo` и `Edit` есть специальные методы для обмена через буфер текстовой информацией. Методы `CopyToClipBoard` и `CutToClipBoard` помещают данные

в буфер обмена, *копируя* и *вырезая* ее из источника соответственно, а метод PasteFromClipBoard *вставляет* текстовый фрагмент из буфера обмена в поле редактирования.

Данные, помещаемые в объект Clipboard или извлекаемые из него с помощью названных методов, по умолчанию имеют текстовый формат. При чтении данных из буфера обмена в элементы редактирования Memo или Edit возможна ситуация, когда эти данные не являются текстом. В этом случае метод PasteFromClipBoard будет работать не так, как ожидалось: поле редактирования не изменится, или в нем появится "мусор".

В приводимой далее процедуре обработки нажатия кнопки происходит копирование содержимого элемента редактирования Memo1 в буфер обмена:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.SelectAll;
  Memo1.CopyToClipboard;
end;
```

При работе с графическими компонентами для операций, связанных с обменом информацией через общую область, удобно использовать метод Assign. Процедура Assign(Source: TPersistent) *присваивает* буферу обмена *объект*, указанный параметром Source. В случае, когда объект принадлежит к графическим классам, например, TBitmap, TPicture или TMetafile, в буфер обмена копируется изображение соответствующего формата (CF_Bitmap или CF_MetaFilePict). Для *извлечения изображения* из объекта Clipboard также можно применить метод Assign.

Рассмотрим в качестве примера следующую процедуру:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  Clipboard.Open;
  Clipboard.Assign(Image1.Picture);
  // В этом месте могут быть расположены инструкции, выполняющие
  // обработку изображения, которое находится в буфере обмена
  if Clipboard.HasFormat(CF_Picture)
    then Image2.Picture.Assign(Clipboard)
    else  Image2.Picture.Assign(nil);
  Clipboard.Close;
end;
```

Изображение, находящееся в компоненте Image1, помещается в буфер обмена, откуда затем копируется в компонент Image2. Для выполнения этих операций устанавливается монопольный доступ к объекту Clipboard. Перед записью изображения из буфера в компонент Image2 могут быть выполнены различные действия, связанные с обработкой изображения.

Как уже говорилось, объект Clipboard представляет собой *глобальный буфер* обмена — общую для всех приложений область оперативной памяти и связанные с ней операции. Кроме глобального, в программе могут использоваться также *локальные буферы обмена*, представляющие собой области оперативной памяти, выделенные программе во время ее выполнения для временного хранения каких-либо данных.

Для *обмена* данными между объектом `Clipboard` и локальным буфером предназначены методы `GetTextBuf` и `SetTextBuf`. Они позволяют осуществлять обмен текстовой информацией между двумя буферами, один из которых является буфером обмена Windows (глобальным буфером), а другой — локальным буфером, выделенным программистом. Прежде методы `GetTextBuf` и `SetTextBuf` использовались часто, т. к. имели ограничения на максимальную длину строки. Начиная с Delphi 4, эти ограничения сняты, и операции *чтения* и *записи* в буфер обмена текстовой информации удобнее выполнять с помощью рассмотренного ранее свойства `AsText` объекта `Clipboard`, а также свойства `Text` и методов `CopyToClipboard`, `CutToClipboard` и `PastFromClipboard` компонентов.

Для *помещения текста* в буфер обмена с использованием дескриптора Windows и для обеспечения доступа к этому тексту предназначены методы `SetAsHandle` и `GetAsHandle`. Эти методы могут применяться при вызове API-функций и на практике используются редко.

Динамический обмен данными

Динамический обмен данными (Dynamic Data Exchange — DDE) представляет собой технологию, связанную с передачей данных между приложениями, работающими под управлением операционной системы Windows. С помощью технологии DDE два приложения могут во время их выполнения динамически взаимодействовать и обмениваться текстовыми данными. При этом изменения в одном приложении немедленно отражаются во втором приложении. Кроме того, с помощью технологии DDE можно из одного приложения управлять другими приложениями, например, Microsoft Word или Excel.

При динамическом обмене два приложения соблюдают соглашение об обмене и устанавливают между собой непосредственную связь на время передачи данных. При этом программа, запрашивающая данные, становится *клиентом*, а программа, служащая источником данных, является *сервером*. В зависимости от направления передачи данных одно и то же приложение может одновременно быть и клиентом, и сервером. Организация динамического обмена данными включает в себя два этапа:

- установление связи между клиентом и сервером (может устанавливаться как при разработке, так и при выполнении приложения);
- собственно передача текстовых данных, при этом возможны следующие действия:
  - получение данных от сервера;
  - передача данных на сервер;
  - посылка серверу команд (макросов).

Delphi позволяет создавать оба типа приложений — сервер и клиент, при этом каждое из них создается отдельно. Одновременную отладку сервера и клиента удобно осуществлять с помощью Менеджера проектов (Project Manager), который позволяет удобно и быстро переключаться между проектами.

Для приложений, реализующих DDE, используются специальные компоненты страницы **System** Палитры компонентов.

Приложение-сервер

Приложение-сервер отвечает за передачу данных клиенту. Для организации обмена в приложении-сервере должны быть размещены два специальных компонента:

- `DdeServerConv` (сеанс обмена данными);
- `DdeServerItem` (передаваемые данные).

Обычно приложение имеет один компонент `DdeServerConv` и несколько компонентов `DdeServerItem`. В процессе передачи данных эти компоненты используются совместно.

Назначением компонента `DdeServerConv` является общее *управление обменом* и *обработка запросов* от клиентов на выполнение макросов.

Компонент `DdeServerItem` через свойство `ServerConv` типа `TDdeServerConv` связан с компонентом `DdeServerConv` и непосредственно *определяет текстовые данные* сервера, пересылаемые в процессе обмена. Эти данные должны быть занесены в следующие свойства:

- `Text` типа `String` (однострочные данные);
- `Lines` типа `TStrings` (многострочные данные).

При изменении значений этих свойств происходит автоматическая пересылка обновленных данных во все приложения клиентов, установившие связь с сервером. Отметим, что свойства `Text` и `Lines[0]` имеют одинаковые значения. Своевременное обновление свойств компонента `TDdeServerItem`, содержащих данные, является обязанностью программиста. Если, например, компонент `TDdeServerItem` берет передаваемый текст из компонента `Edit`, то обновление свойства `Text` может быть выполнено в обработчике события `OnChange` этого компонента с помощью следующей инструкции:

```
DdeServerItem1.Text := Edit1.Text;
```

Если передается многострочный текст, то используется свойство `Lines`, а текст может быть взят, например, из компонента `Memo`.

При подключении к серверу клиента последнему нужна информация о параметрах сервера. Эти параметры можно поместить в буфер обмена с помощью метода `CopyToClipboard` компонента `DdeServerItem`. Впоследствии эти параметры можно извлечь и использовать в приложении-клиенте для осуществления подключения к серверу.

В случае, если сервер может принимать данные от клиента, программист должен подготовить обработчик события `OnPokeData` типа `TNotifyEvent` компонента `DdeServerItem`. Например, при отображении получаемых данных в компоненте `Edit2` в обработчике может присутствовать инструкция:

```
Edit2.Text := DdeServerItem1.Text;
```

Если сервер должен *обрабатывать макросы*, то соответствующие действия кодируются в обработчике события `OnExecuteMacro` типа `TMacroEvent` компонента `DdeServerConv`. Тип события описан следующим образом:

```
type TMacroEvent = procedure(Sender: TObject; Msg: TStrings) of object;
```

Рассмотрим в качестве примера приложение-сервер (рис. 37.1), которое может передавать клиенту и получать от него данные. Для передачи данных используется компонент

edtServerPut типа TEdit, данные автоматически передаются в процессе их изменения. Получение данных в компонент edtServerGet происходит при их посылке из приложения-клиента методами PokeDataLines и PokeDataLines.

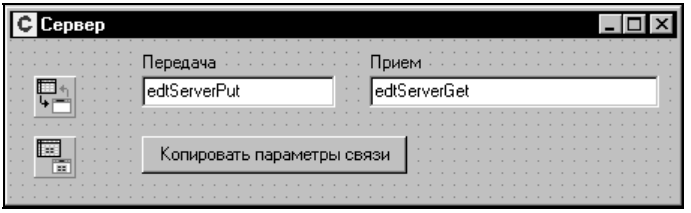


Рис. 37.1. Форма приложения сервера при разработке

В листинге 37.1 приведен код модуля uServer формы Form1 приложения-сервера.

Листинг 37.1. Приложение-сервер для передачи и получения данных

```
unit uServer;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, DdeMan;
type
  TForm1 = class(TForm)
    DdeServerConv1: TDdeServerConv;
    DdeServerItem1: TDdeServerItem;
      edtServerPut: TEdit;
      btnPasteLink: TButton;
      edtServerGet: TEdit;
            Label1: TLabel;
            Label2: TLabel;
    procedure btnPasteLinkClick(Sender: TObject);
    procedure edtServerPutChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure DdeServerItem1PokeData(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}

// Установление связи между компонентами
// DdeServerItem1 и DdeServerConv1
procedure TForm1.FormCreate(Sender: TObject);
begin
  DdeServerItem1.ServerConv := DdeServerConv1;
end;
```

```
// Копирование параметров сервера в буфер обмена
procedure TForm1.btnPasteLinkClick(Sender: TObject);
begin
  DdeServerItem1.CopyToClipboard;
end;

// Текстовые данные компонента DdeServerItem1 устанавливаются
// при изменении содержимого компонента edtServerPut
procedure TForm1.edtServerPutChange(Sender: TObject);
begin
  DdeServerItem1.Text := edtServerPut.Text;
end;

// Текстовые данные компонента edtServerGet
// устанавливаются при их получении от клиента
procedure TForm1.DdeServerItem1PokeData(Sender: TObject);
begin
  edtServerGet.Text := DdeServerItem1.Text;
end;
end.
```

Для организации сеанса связи и передачи данных в форме приложения размещены связанные между собой компоненты `DdeServerConv1` и `DdeServerItem1`. Для копирования в буфер обмена параметров сервера, требуемых для подключения клиентов, используется кнопка `btnPasteLink`.

Приложение-клиент

Приложение-клиент отвечает за *получение данных* от сервера. Для организации обмена в приложении-клиенте должны быть размещены два специальных компонента:

- `DdeClientConv` (сеанс обмена данными);
- `DdeClientItem` (получаемые данные).

Обычно приложение имеет один компонент `DdeClientConv` и несколько компонентов `DdeClientItem`. В процессе передачи данных эти компоненты используются совместно.

Компонент `DdeClientConv` обеспечивает *установление сеанса* связи с сервером и *управление процессом обмена* данными, для чего параметры сервера заносятся в следующие свойства (типа `String`) компонента:

- `DDEService` — имя сервера (имя исполняемого файла приложения-сервера без расширения exe);
- `DDETopic` — предмет обмена; если приложение-сервер создано в Delphi, то предметом обмена является компонент `DdeServerConv`, и необходимо указать его имя;
- `ServiceApplication` — полное имя исполняемого файла приложения-сервера без расширения exe (устанавливается в случае размещения приложения-сервера не в текущем каталоге).

Для приложения-сервера, созданного не с помощью Delphi, требуемые для соединения данные можно узнать из документации к серверу.

Установить значения свойств `DDEService` и `DDETopic` можно в процессе выполнения приложения или через Инспектор объектов. При выполнении приложения значения этих свойств задаются, например, с помощью следующих инструкций присваивания:

```
DdeClientConv1.DdeService := 'ProjectServer';
DdeClientConv1.DdeTopic := 'DdeServerConv1';
```

При разработке приложения параметры сервера можно ввести через Инспектор объектов. Более удобно использовать окно **DDE Info** (рис. 37.2), которое вызывается из области значения свойств `DDEService` и `DDETopic`. После нажатия кнопки **Paste Link** (Вставить параметры соединения) параметры сервера из буфера обмена заносятся в соответствующие свойства. Предварительно эти параметры нужно скопировать в буфер обмена методом `CopyToClipboard`. В рассмотренном выше приложении-сервере это выполнялось при нажатии кнопки **Копировать параметры связи** (`btnPasteLink`).

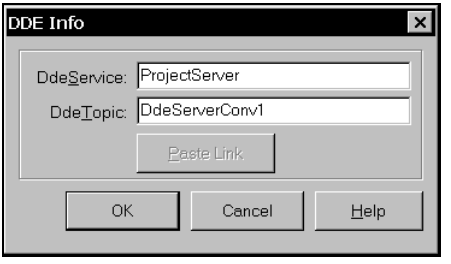


Рис. 37.2. Окно информации DDE

Свойство `ConnectMode` типа `TDataMode` компонента `DdeClientConv` управляет *режимом подключения* клиента к серверу.

Это свойство может принимать следующие значения:

- `ddeAutomatic` (автоматическое соединение);
- `ddeManual` (ручное соединение).

По умолчанию свойство `ConnectMode` установлено в значение `ddeAutomatic`, и при запуске клиента соединение с сервером происходит автоматически. Если при запуске приложения-клиента приложение-сервер еще не было запущено, то оно автоматически запускается.

Программист может самостоятельно *управлять подключением клиента* к серверу, для этого он должен установить свойство `ConnectMode` в значение `ddeManual` и вызвать метод `OpenLink`.

Пример подключения к серверу:

```
if DdeClientConv1.ConnectMode <> ddeAutomatic then
     DdeClientConv1.OpenLink;
```

При выполнении приложения-клиента *параметры соединения* с сервером можно установить методом `SetLink`. Функция `SetLink(Service: String; Topic: String): Boolean` указывает сервер, который задается параметрами `Service` (имя сервера) и `Topic` (предмет обмена), а также устанавливает с ним связь, если свойство `ConnectMode` имеет значение `ddeAutomatic`.

Например, следующий код реализует подключение к серверу:

```
if DdeClientConv1.SetLink('ProjectServer', ' DdeServerConv1') then
  begin
    Beep;
    DdeClientItem1.DdeConv:=DdeClientConv1;
    DdeClientItem1.DdeItem:='DdeServerItem1';
    if  DdeClientConv1.ConnectMode <> ddeAutomatic
        then DdeClientConv1.OpenLink;
  end;
```

Для отключения от сервера используется метод `CloseLink`.

Компонент `DdeClientItem` *определяет текстовые данные* клиента, пересылаемые в процессе обмена. Он связан через свойство `DDEConv` типа `TDdeServerConv` с компонентом `DdeClientConv`, а через свойство `DDEItem` типа `String` — с находящимся на сервере компонентом `DdeServerItem`. Текстовые данные компонента `DdeClientItem` содержатся в его свойствах:

- `Text` типа `String` (однострочные данные);
- `Lines` типа `TStrings` (многострочные данные).

Эти свойства не отличаются от аналогичных свойств компонента `DdeServerItem` сервера. При изменении данных компонента `DdeClientItem` генерируется событие `OnChange` типа `TNotifyEvent`, обработчик которого используется для приема этих данных. Например, если поступающий текст должен отображаться в компоненте `Edit1`, то в обработчике можно разместить следующую инструкцию:

```
Edit1.Text := DdeClientItem1.Text;
```

Клиент может принимать данные от сервера и посылать ему данные, а также команды для выполнения (макросы). Для этого используются описанные далее методы компонента `DdeClientConv`.

Функция `PokeData(Item: String; Data: PChar): Boolean` посылает на сервер строку текста, указанную параметром `Data`. Так как параметр `Data` имеет тип `PChar`, передаваемая строка должна быть преобразована к этому типу. Параметр `Item` специфицирует элемент данных. Функция возвращает логическое значение, указывающее на успешность выполнения операции.

Пример посылки данных на сервер:

```
var pc: array[0..100] of Char;
...
DdeClientConv1.PokeData(DdeClientItem1.DDEItem,
    StrPCopy(pc, edtClientPut.Text));
```

Здесь для преобразования передаваемой строки (до 100 символов) к типу `PChar` применены функция `StrPCopy` и промежуточная переменная `pc`. Результат передачи данных не анализируется.

Для *передачи нескольких строк* используется функция `PokeDataLines(Item: String; Data: TStrings): Boolean`.

Для *отображения* и *обработки* принимаемых данных на сервере необходимо подготовить обработчик события `OnPokeData` компонента `DdeServerItem`.

Приложение-клиент может посылать на сервер *макросы* — команды, которые могут быть выполнены на сервере. *Посылка макросов* осуществляется методами `ExecuteMacro` и `ExecuteMacroLines`. *Выполнение макросов* на сервере кодируется в обработчике события `OnExecuteMacro` компонента `DdeServerConv`. Состав макросов и их особенности зависят от конкретных приложений-серверов и описываются в сопроводительной документации.

В качестве примера рассмотрим приложение-клиент (рис. 37.3), которое при старте автоматически запускает приложение-сервер (если оно еще не запущено). В качестве сервера использовано приложение, приведенное в предыдущем разделе. Параметры соединения и свойства компонента `DdeClientItem1` устанавливаются при создании формы. Обычно это выполняется через Инспектор объектов.

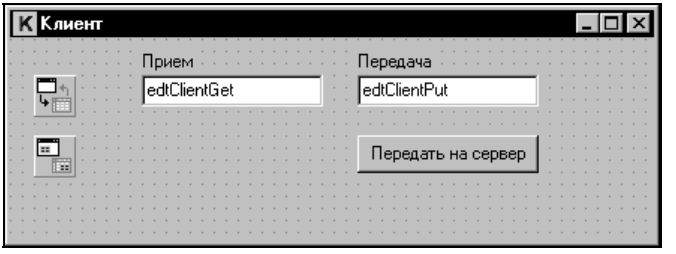


Рис. 37.3. Форма приложения-клиента при разработке

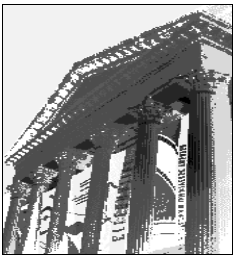
В листинге 37.2 приведен код модуля `uClient` формы `Form1` приложения-клиента.

Листинг 37.2. Пример приложения-клиента

```
unit uClient;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, DdeMan, StdCtrls;
type
  TForm1 = class(TForm)
      edtClientGet: TEdit;
    DdeClientConv1: TDdeClientConv;
    DdeClientItem1: TDdeClientItem;
      edtClientPut: TEdit;
            btnPut: TButton;
            Label1: TLabel;
            Label2: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure DdeClientItem1Change(Sender: TObject);
    procedure btnPutClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  // Указание сеанса связи
  DdeClientItem1.DDEConv := DdeClientConv1;
  // Задание источника данных
  DdeClientItem1.DdeItem := 'DdeServerItem1';
  // Установка параметров соединения
  DdeClientConv1.DdeService := 'ProjectServer';
  DdeClientConv1.DdeTopic := 'DdeServerConv1';
end;

// Копирование в компонент-редактор данных, получаемых от сервера
procedure TForm1.DdeClientItem1Change(Sender: TObject);
begin
  edtClientGet.Text := DdeClientItem1.Text;
end;

// Посылка данных (до 100 символов) из компонента-редактора на сервер
procedure TForm1.btnPutClick(Sender: TObject);
var pc: array[0..100] of Char;
begin
  DdeClientConv1.PokeData(DdeClientItem1.DDEItem,
    StrPCopy(pc, edtClientPut.Text));
end;
end.
```

Поскольку компонент `DdeClientItem1` через свойство `DdeItem` связан с компонентом `DdeServerItem1`, при изменении данных последнего эти изменения отражаются в компоненте `DdeClientItem1`. Чтобы отобразить эти данные в элементе редактирования `edtClientGet`, создан обработчик события `OnChange`. Посылка данных на сервер выполняется при нажатии кнопки `btnPut`, данные берутся из компонента `edtClientPut`.



ГЛАВА 38

Подготовка приложения к распространению

Профессионально разработанное приложение должно быть снабжено справочной системой и предоставлять возможность его автоматизированной инсталляции на компьютере пользователя. Оба этих аспекта подробно освещены в данной главе.

Создание справочной системы

Для этого в его состав должен входить справочный файл с расширением hlp, который автоматически вызывается при нажатии клавиши <F1> или при выборе соответствующего пункта меню, например, **Справка** (Help). При запуске справочной системы автоматически выполняется программа winhelp.exe или winhlp32.exe, главное окно которой показано на рис. 38.1. В это окно загружается соответствующий справочный файл.

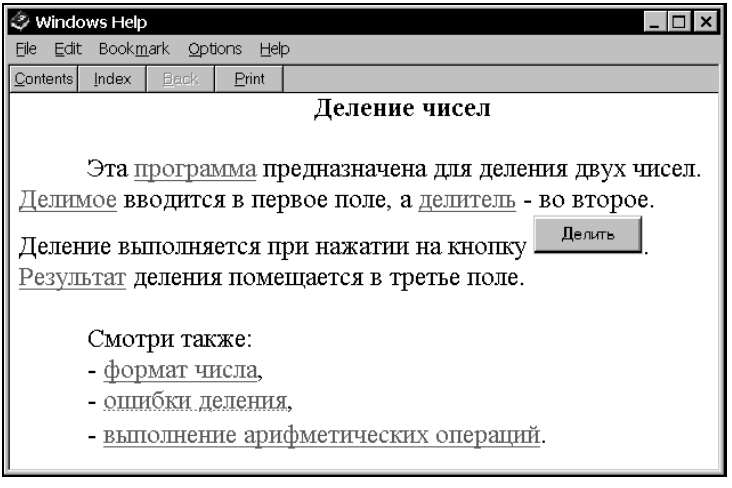


Рис. 38.1. Окно справки Windows

В окне справки есть меню и панель инструментов, позволяющие работать со справочными файлами.

С помощью команд меню **File** можно открыть новый справочный файл, выбрать принтер и распечатать содержимое текущего раздела, отображаемого в главном окне справки. Распечатать можно отдельный раздел, а не весь справочный файл.

Команды меню **Edit** позволяют копировать содержимое текущего раздела или его фрагмент в буфер обмена, а также вставлять комментарии к разделу.

Команды меню **Bookmark** предназначены для определения закладок и последующего перехода на них.

Команды меню **Options** позволяют установить отдельные параметры окна справки, например, шрифт, цвет окна, размещение на экране.

Кнопки панели инструментов имеют следующее назначение:

- **Contents** — вывод раздела, содержащего оглавление справочного файла;
- **Index** — вывод окна поиска (предметного указателя);
- **Back** — возврат к разделу, который отображался в окне перед текущим разделом;
- **Print** — печать раздела.

В зависимости от ситуации отдельные кнопки могут быть неактивны. Кроме того, разработчик справочной системы с помощью макрокоманд может изменить состав меню и панели инструментов.

Помимо основного окна справки, можно создать вторичные перекрывающиеся и временные окна. В них также отображается содержимое разделов справочной системы.

В организации справочной системы приложения можно выделить следующие четыре этапа:

1. Определение номеров контекста справочной системы для элементов управления (компонентов) формы.
2. Создание текста файла справки с помощью текстового редактора.
3. Создание справочного файла с помощью специального компилятора.
4. Подключение справочных файлов к приложению и организация их вызова.

Операции первого и четвертого этапа выполняются в среде Delphi.

Справочный контекст компонента

Каждый элемент управления (компонент) формы, а также сама форма могут иметь *номер контекста* для справочной системы. *Контекст* представляет собой число, задаваемое свойством `HelpContext`, по значению которого определяется раздел справочной системы, соответствующий данному элементу. При выполнении приложения, если элемент управления находится в фокусе ввода, нажатие клавиши <F1> приводит к отображению на экране раздела контекстной справки для этого элемента. Если раздел справки не был подготовлен или произошла какая-либо ошибка, то выдается соответствующее сообщение.

Текстовый файл справки

Текстовый файл справки создается как документ в формате RTF. Для его подготовки можно использовать любой редактор, позволяющий работать с документами этого формата, в том числе текстовый процессор Microsoft Word. Этот процессор позволяет создавать для справочной системы текстовый файл в режиме "почти WYSIWYG", когда внешний вид редактируемого документа почти совпадает с тем, что увидит пользователь, запустивший справочную систему. В частности, при отображении справки в ссылках не показываются контексты разделов, а текст ссылок выделяется подчеркиванием и цветом. Дальнейший материал излагается для Microsoft Word 2000.

Документ может содержать графические изображения и таблицы, стилевое и шрифтовое оформление. Кроме собственно справочной информации, в него включаются управляющие данные, например, определяющие взаимосвязи между разделами документа.

Основным элементом справочной системы является раздел (страница). *Раздел* представляет собой фрагмент справочной системы, отображаемый в окне приложения winhelp.exe. Он может содержать текст и графические изображения. Если размер окна недостаточен для отображения раздела целиком, у окна появляется горизонтальная или вертикальная полоса прокрутки. Каждый раздел справки оформляется отдельным разделом документа.

Текст справочной системы представляется в виде *гипертекста* и состоит из многих разделов, связанных между собой *перекрестными ссылками*. Каждый раздел обычно содержит заголовок, отображаемый в верхней части окна просмотра, идентификатор, набор ключевых слов, по которым можно найти раздел, а также ссылки на другие разделы.

Справочный файл имеет оглавление, или начальный раздел, который при иерархической организации справки играет роль корневого раздела.

Раздел справочного файла может иметь следующие атрибуты:

- контекст — текстовая строка, однозначно идентифицирующая раздел;
- заголовок — название, под которым раздел появляется при поиске в окне справки; максимальная длина заголовка составляет 128 символов;
- список ключевых слов — слова для поиска в окне справки, максимальная длина одного слова составляет 255 символов; если список содержит несколько ключевых слов, то они разделяются точкой с запятой (`;`);
- номер в последовательности просмотра — положение раздела при последовательном просмотре справочного файла;
- макрокоманда — выполняется при отображении раздела;
- тег компиляции — признак, позволяющий включать или не включать раздел в справочную систему в зависимости от параметров компиляции.

Из всех атрибутов обязательным является *контекст*, представляющий собой уникальный в пределах файла идентификатор раздела. Для обозначения контекста можно ис-

пользовать русские и латинские буквы, цифры, знак подчеркивания. Контекст может начинаться с любого символа, в том числе с цифры, а его максимальная длина составляет 255 символов.

Атрибуты назначаются разделу с помощью сносок, представляющих собой текст, который располагается в нижней части страницы и может отделяться от основного текста горизонтальной чертой. В тексте сноска оформляется в виде числа или специального символа, например #.

Для вставки сноски следует установить курсор слева или справа от названия раздела и выполнить команду **Insert | Footnote** (Вставка | Сноска), в результате чего появляется диалоговое окно **Footnote and Endnote** (Сноски), показанное на рис. 38.2.

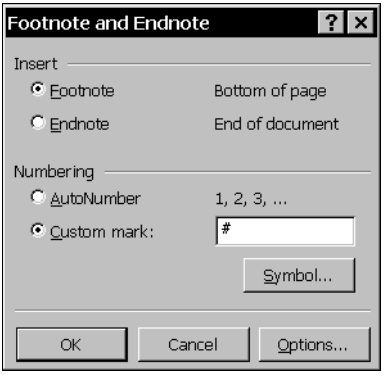


Рис. 38.2. Вставка сноски

В группе **Numbering** (Нумерация) следует выбрать переключатель **Custom mark** (Другая) и ввести в поле символ, соответствующий указанному в скобках атрибуту, который назначается разделу:

- # (контекст);
- $ (заголовок);
- K (список ключевых слов);
- + (номер в последовательности просмотра);
- ! (макрокоманда);
- * (тег компиляции).

После вставки сноски документ оказывается разделенным на две части — в нижней части появляется окно сносок. Текст сноски, который для справочного файла представляет собой атрибут раздела, вводится после соответствующего символа в окне сносок. Между символом сноски и текстом должен быть один символ пробела. Если окно сносок закрыто, то его можно вывести командой **View | Footnotes** (Вид | Сноски), которая доступна только в случае, если для документа создана хотя бы одна сноска.

Для навигации (переходов между разделами) по справочной системе отдельные разделы связаны между собой с помощью *ссылок*.

Можно задать ссылку для перехода:

- на другой раздел;
- во временное окно;
- на раздел другого справочного файла.

Внешне ссылки представляются в виде выделенного цветом и подчеркиванием текста или в виде значков. Внешний вид ссылок зависит от настроек Windows, для стандартной схемы текст ссылки подчеркнут и отображается зеленым цветом. Мы будем предполагать, что установлена стандартная схема.

Разработчик оформляет ссылки в документе специальным образом. Ссылки создаются непосредственно в тексте раздела, для чего используется соответствующее шрифтовое оформление:

- перечеркнутый или подчеркнутый двойной линией текст;
- подчеркнутый одной линией текст;
- скрытый текст.

Ссылка состоит из двух частей. Первая часть — это текст, который виден пользователю и выделен подчеркиванием и цветом. Для выполнения перехода на раздел, обозначенный ссылкой, пользователь должен щелкнуть на выделенном тексте. Вторая часть не видна пользователю и представляет собой строку контекста раздела, на который выполняется переход.

Если выполняется переход на обычный раздел, то первая часть ссылки оформляется как *перечеркнутая* или *подчеркнутая двойной линией* строка текста. При выполнении перехода во временное справочное окно используется *подчеркивание одной линией*. Строка контекста оформляется скрытым текстом. Оформлять символ конца строки как невидимый нежелательно, т. к. в этом случае строка сливается со следующей строкой.

При создании ссылки на раздел можно указать, что его содержимое должно отображаться во вторичном окне. Для этого после контекста ставятся символ `>` и имя окна.

Можно создать ссылку на раздел, расположенный в другом справочном файле. Для этого после контекста указываются символ `@` и имя справочного файла, которые тоже оформляются скрытым текстом. Раздел другого справочного файла также может быть показан во временном или вторичном окнах.

Замечание

Между всеми элементами ссылки не должно быть пробелов.

Чтобы контекст, оформленный скрытым текстом, был виден в документе, на странице **View** (Вид) окна параметров документа следует установить флажок **Hidden text** (Скрытый текст).

Для оформления символов подчеркиванием или скрытым текстом можно использовать окно **Font** (Шрифт), открываемое командой **Format | Font** (Формат | Шрифт).

Рассмотрим пример, в котором реализуется оформление ссылки на раздел. Пусть первая строка представляет собой ссылку на обычный раздел с контекстом `Format`. Вторая строка будет ссылкой на временное окно справки, в котором выводится раздел с контекстом `Errors`. В третьей строке находится ссылка на раздел с контекстом `FindError`,

который отображается во вторичном окне с именем `WindowsFE`. Имя вторичного окна должно быть определено при компиляции файла. Четвертая строка представляет собой ссылку на раздел с контекстом `19` справочного файла D:\BOOK_D7\HELP \operations.hlp. Подготовленный в Microsoft Word 2000 текст будет выглядеть так:

Смотри также:
- ~~формат числа~~Format,
- ошибки деленияErrors,
- ~~поиск и устранение ошибок~~FindError>WindowFE,
- ~~выполнение арифметических операций~~*19*@D:\BOOK_D7\HELP\operations.hlp.

Текстовый документ может содержать рисунки, которые вставляются обычным путем. При этом рисунки, как и текст, можно использовать для ссылки на другой раздел. Для этого рисунок соответствующим образом подчеркивается или перечеркивается, а контекст раздела размещается сразу после рисунка.

После подготовки всех разделов справки документ сохраняется как файл формата RTF.

Создание справочного файла

Для использования в приложении текст справки должен быть преобразован из формата RTF в формат HLP, при этом в процессе преобразования также учитываются номера контекстов элементов управления. Такое преобразование из формата в формат называется *компиляцией справочного файла* и выполняется с помощью специальных программ (довольно многочисленных). Мы рассмотрим создание справочного файла на примере использования программы Microsoft Help Workshop (Компилятор справочных файлов), исполняемый файл hcrtf.exe которой находится в каталоге HELP\TOOLS каталога Delphi. Главное окно компилятора, в котором открыт проект divide.hpj, показано на рис. 38.3.

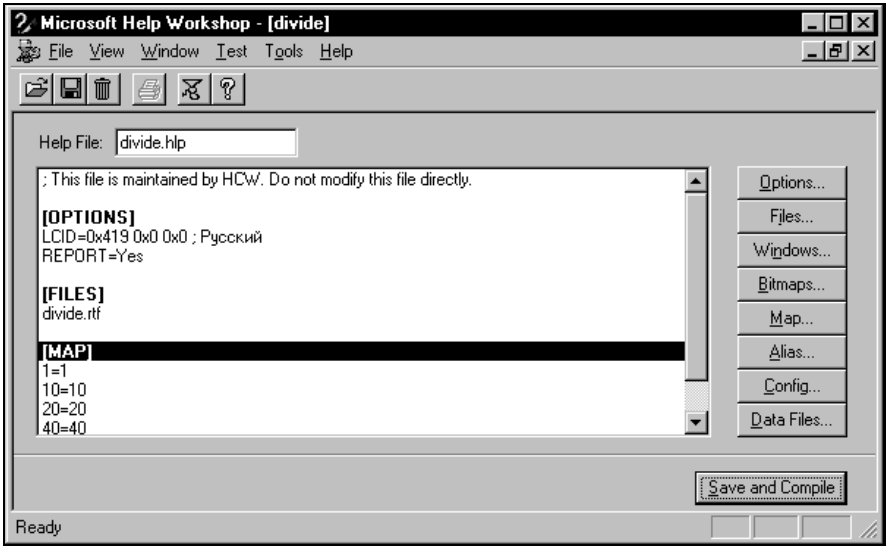


Рис. 38.3. Компилятор справочных файлов

Основным файлом компилятора является файл проекта, который объединяет такие элементы, как текстовые файлы справки, опции, номера контекстов, и позволяет создать из них справочный файл. Создание нового файла справочного проекта выполняется командой **File | New**, в результате чего появляется запрос, показанный на рис. 38.4.

После выбора объекта **Help Project** (Справочный проект) и нажатия кнопки **OK** открывается диалоговое окно выбора имени файла для сохранения, в котором необходимо указать каталог и имя файла проекта. Файлу проекта присваивается расширение hpj.

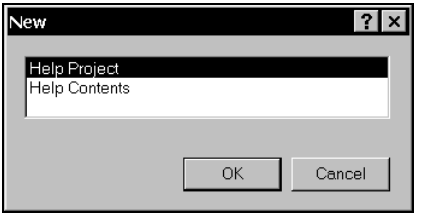


Рис. 38.4. Окно запроса на создание нового справочного проекта

Элементы проекта представляют собой отдельные секции, имена которых отображаются заглавными буквами и заключены в квадратные скобки. Первоначально в проекте есть только секция `OPTIONS`, в которую включены параметры окна справки и вывода отчета о компиляции проекта. Кроме этих, есть много других параметров, управляющих процессом создания справочного файла. Однако для создания большинства справочных файлов знать их не обязательно, поэтому здесь они не рассматриваются.

Список текстовых файлов, на основании которых строится справочный файл, содержится в секции `FILES`, в приведенном примере это divide.rtf. Для управления содержимым этой секции следует нажать кнопку **Files**, после чего открывается окно **Topic Files** (рис. 38.5), содержащее список подключенных текстовых файлов. В этом окне можно добавлять и удалять файлы.

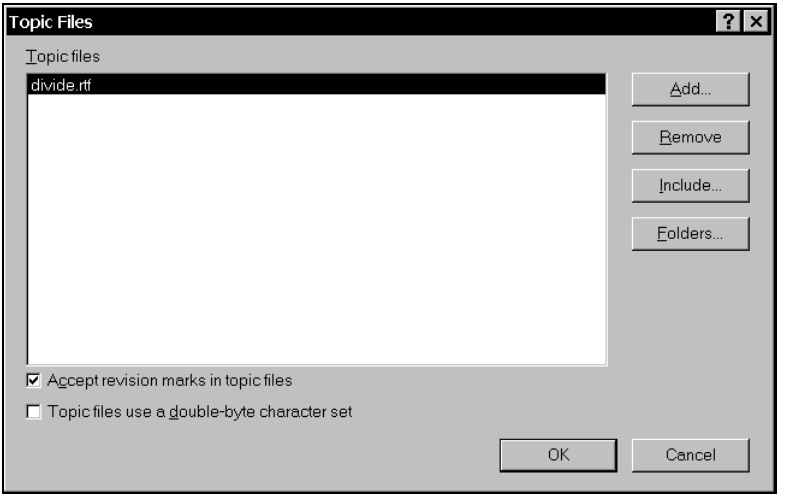


Рис. 38.5. Текстовые файлы справки

Для добавления файла следует нажать кнопку **Add** и в появившемся окне выбрать нужный файл формата RTF. Нажатие кнопки **Remove** удаляет выбранный файл из списка.

Обычно при вызове справки в процессе работы приложения автоматически вызывается раздел, соответствующий элементу управления, который находится в фокусе ввода. Чтобы это происходило, нужно установить соответствие между номерами контекстов элементов управления приложения и разделами справочной системы. Карта соответствия номеров контекстов и разделов справочной системы указывается в секции `MAP` проекта. Для изменения этой карты нажатием кнопки **Map** открывается одноименное диалоговое окно (рис. 38.6).

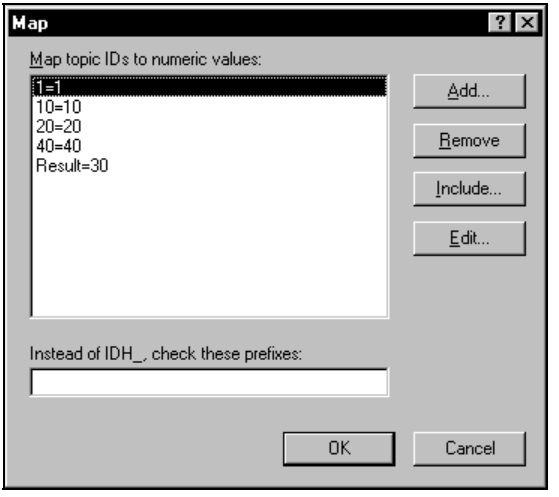


Рис. 38.6. Карта соответствия номеров контекстов и разделов

В этом окне разработчик может добавлять новые соответствия или удалять существующие. Соответствия отображаются построчно в виде пар значений, разделенных знаком равенства. Слева находится контекст раздела справочного файла (сноска, обозначенная символом `#`), а справа — номер контекста элемента управления приложения.

При нажатии кнопки **Save and Compile** выполняется компиляция проекта и создается справочный файл, имя которого совпадает с именем файла проекта. Результаты компиляции выдаются в специальном окне и имеют вид:

```
Creating the help file divide.hlp
Processing D:\Book_d7_Examples\Help\divide.rtf
   HC3104: Warning: topic #7 of D:\Book_d7_Examples\Help\divide.rtf :
      The Topic ID footnote (#) does not specify a Topic ID.
   HC3025: Warning: topic #1 of D:\Book_d7_Examples\Help\divide.rtf :
      Jump to undefined Topic ID: "Format".
Resolving keywords...
Adding bitmaps...
7      Topics
12      Jumps
4      Keywords
1      Bitmap
```

```
Created D:\Book_d7_Examples\Help\divide.hlp, 9,653 bytes
Bitmaps: 514 bytes
Compile time: 0 minutes, 0 seconds
0 notes, 2 warnings
```

Если сообщений об ошибках не было, то справочный файл содержит корректную информацию и готов к использованию.

Замечание

Если при компиляции были выявлены ошибки, то справочный файл все равно создается и может быть использован, однако отдельные его элементы, например ссылки, могут не работать или работать неправильно.

Компилятор справочных файлов является MDI-приложением и может иметь несколько одновременно открытых дочерних окон. После выполнения компиляции такими окнами являются окно проекта и окно результатов компиляции, причем последнее располагается поверх окна проекта. Для продолжения работы над проектом необходимо закрыть окно результатов или переключиться в окно проекта командой **Window | *Project Name***, где ***Project Name*** — имя файла проекта.

Для проверки созданного справочного файла его можно загрузить и просмотреть в окне справки Windows (см. рис. 38.1).

Подключение справочного файла

Справочный файл, разработанный для использования в составе приложения, следует подключить к приложению. На этапе разработки имя справочного файла приложения можно указать в поле **Help file** страницы **Application** окна параметров проекта. В результате при запуске приложения указанный файл будет подключен автоматически. Подключать и заменять справочный файл можно также динамически с помощью инструкции вида

```
Application.HelpFile := <Имя справочного файла>;
```

После подключения к приложению справочного файла его разделы автоматически вызываются при нажатии клавиши <F1> на тех элементах управления, которые через свойства `HelpContext` связаны с этими разделами.

Если меню приложения содержит команду **Помощь** или подобную ей, то для вызова справки нужно подготовить соответствующий обработчик. Этот обработчик может содержать следующий код:

```
Application.HelpCommand(Help_Contents, 0);
```

В результате вызывается начальный раздел подключенного файла справочной системы.

Пример создания справочной системы

Рассмотрим создание справочной системы на примере программы, выполняющей деление двух чисел. Вид формы этого приложения на этапе разработки показан на рис. 38.7.

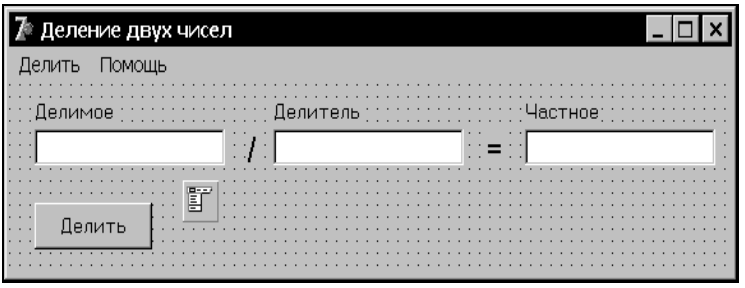


Рис. 38.7. Форма приложения для деления двух чисел при разработке

Приведем названия, назначение и номера контекстов элементов (значение свойства `HelpContext`) справочной системы элементов управления:

- `Edit1` (делимое) — `10`;
- `Edit2` (делитель) — `20`;
- `Edit3` (результат) — `30`;
- `Button1` (выполнение деления) — `40`.

Для приложения создан файл в формате RTF, текст которого приведен далее. При просмотре RTF-файла в текстовом процессоре Microsoft Word вид файла будет иным, в первую очередь это относится к сноскам, текст которых выводится в отдельном окне. Для наглядности в данном примере текст сносок сгруппирован по разделам и выделен курсивом.

$ K # **Деление чисел**

Эта ~~программа~~2F предназначена для деления двух чисел. ~~Делимое~~10 вводится в первое поле, а ~~делитель~~20 — во второе. Деление выполняется при нажатии кнопки ~~Делить~~. ~~Результат~~Result деления помещается в третье поле.

Смотри также:
- ~~формат числа~~Format,
- ошибки деленияErrors,
- ~~выполнение арифметических операций~~19@D:\BOOK_D7\HELP\operation.hlp.

$ Деление чисел
K Деление чисел
 # 1

----------Разрыв страницы----------

$ K # **Поле делимого**

В это поле вводится вещественное число, которое является делимым. Число должно вводиться в соответствующем ~~формате~~Format.

Смотри также поле ~~делителя~~20.

$ *Поле делимого*
K *Поле делимого*
10
----------Разрыв страницы----------
$ K # **Поле делителя**

В это поле вводится вещественное число, которое является делителем. Число должно вводиться в соответствующем ~~формате~~Format.
Смотри также поле ~~делимого~~10.

$ *Поле делителя*
K *Поле делителя*
20
----------Разрыв страницы----------
$ K # **Поле результата**

В это поле помещается результат деления. При возникновении ~~ошибки~~Errors деления вместо результата в поле заносится текст "Ошибка!".

$ *Поле результата*
K *Поле результата*
Result
----------Разрыв страницы----------
Выполнение деления

Деление выполняется при нажатии кнопки "Делить".

40
----------Разрыв страницы----------
Ошибки деления

Ошибки деления могут возникнуть при:

- вводе чисел в неправильном формате;
- вводе делителя, равного нулю.

Errors
----------Разрыв страницы----------

Формат числа

Число может быть представлено в виде с фиксированной или плавающей точкой.

Formats

----------Разрыв страницы----------

Для контекстов разделов справочного файла и номеров контекста элементов управления в компиляторе установлены следующие соответствия:

```
10 = 10
20 = 20
40 = 40
Result = 30
```

Отметим, что для ссылки на раздел `Деление чисел` с контекстом `40` использован рисунок с изображением кнопки деления. Рисунок встроен в документ и перечеркнут как обычная ссылка.

Создание дистрибутива приложения

Разработанное программистом приложение в общем случае может быть достаточно сложным и включать в свой состав большое количество файлов, размещаемых в различных каталогах. При инсталляции такого приложения пользователь должен создать на диске требуемую структуру каталогов и скопировать в них соответствующие файлы приложения. Кроме того, в главное меню Windows требуется включить команду запуска приложения и внести изменения в системный реестр. Эти действия подготовленный пользователь может выполнить самостоятельно, проведя достаточно кропотливую работу в соответствии с инструкцией по инсталляции приложения.

Для автоматизации процесса инсталляции приложения на компьютере пользователя обычно создается дистрибутивный (дистрибутив), или инсталляционный, вариант приложения. Для инсталляции приложения с помощью дистрибутива пользователю достаточно запустить программу инсталляции, которая обычно находится на первом диске или в главном каталоге и называется setup.exe или install.exe. Программа инсталляции запускает Мастер инсталляции приложения, который в интерактивном режиме обеспечивает перенос необходимых файлов приложения в нужные каталоги и соответствующую настройку Windows.

Для создания дистрибутива приложения есть специальные программы-утилиты, среди которых отметим InstallShield Express. Эта программа поставляется совместно с Delphi, хотя и устанавливается отдельно. Запустить программу InstallShield Express можно через главное меню Windows или через исполняемый файл iside.exe.

Утилита InstallShield Express облегчает процесс распространения программного продукта, предоставляя следующий сервис:

- настройки по умолчанию, соответствующие установкам Windows для независимых программных продуктов;

- настраиваемый интерфейс пользователя, включающий рисунки и доски объявлений;
- возможность инсталляции дистрибутива с различных устройств, а также через сеть;
- защита дистрибутива с помощью серийного номера;
- автоматическое сжатие файлов;
- поддержка различных языков;
- автоматическое изменение системного реестра и инициализационных файлов;
- включение в состав дистрибутива модулей Windows.

После запуска программы появляется окно **InstallShield Express Borland Limited Edition**, в котором разработчику предлагается выбрать или создать рабочий документ для дальнейшей работы. Рабочим документом утилиты InstallShield Express является проект, имя файла которого имеет расширение ism. Имя этого файла может совпадать с именем файла проекта приложения Delphi, дистрибутив которого создается.

Создание нового проекта начинается командой **Create a new project**, при этом в правой части окна (рис. 38.8) появляются элементы управления, позволяющие выбрать тип проекта (панель **Project Type**), его расположение и имя (поле **Project Name and Location**). Перед началом создания проекта нажатием кнопки **Create** нужно подтвердить его выбранный тип — создание проекта с помощью мастера (**Project Wizard**) или пустой проект (**Blank Setup Project**).

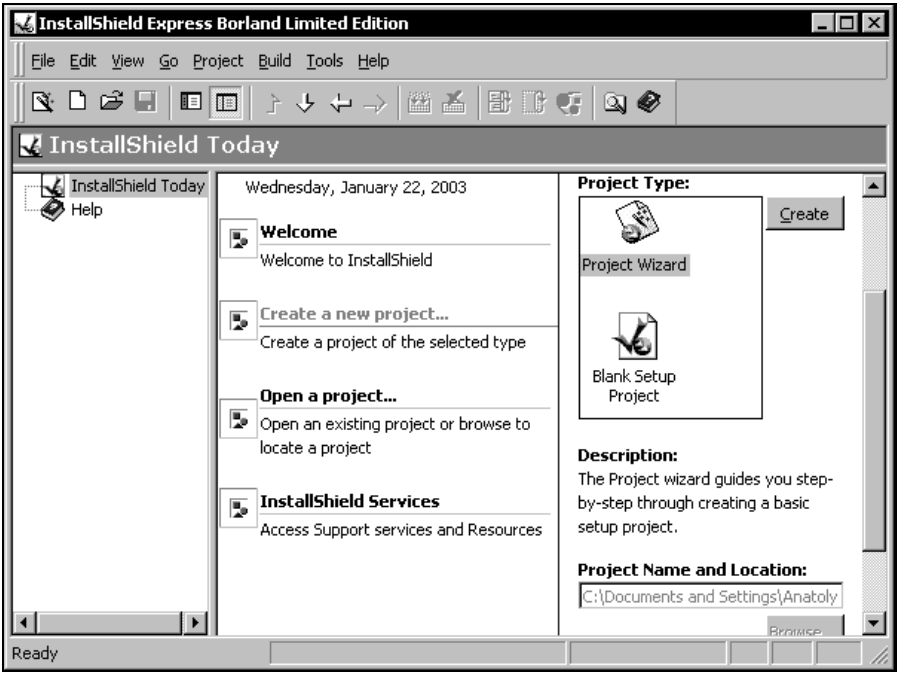


Рис. 38.8. Создание нового проекта

Для открытия имеющегося проекта выбирается команда **Open a project** в средней части окна, при этом в правой части окна (рис. 38.9) выводится список проектов (**Project**

List), а также информация о выбранном в этом списке проекте: имя файла (**File Name**), тип (**Project Type**), дата последнего изменения (**Modified**) и размещение проекта (**Location**). Нажатие кнопки **Open** приводит к открытию проекта, выбранного в списке **Project List**. Если проект отсутствует в списке, то его можно найти, нажав кнопку **Browse** и выбрав требуемый проект в открывшемся окне.

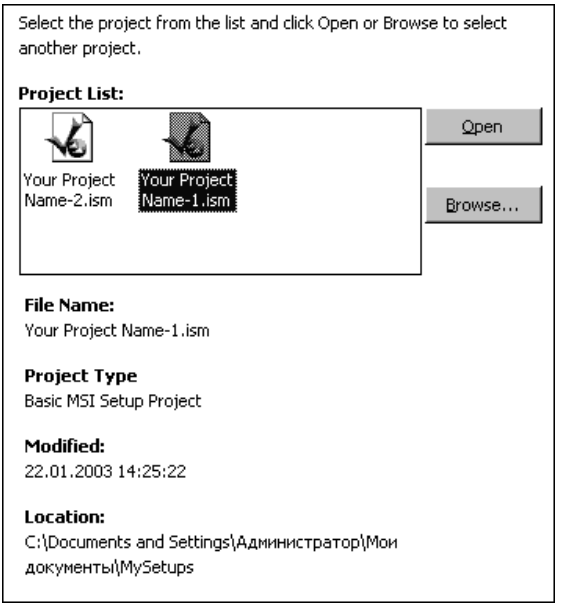


Рис. 38.9. Открытие проекта

После создания нового или открытия существующего проекта дистрибутива окно программы InstallShield Express принимает вид, показанный на рис. 38.10. Отметим, что программа InstallShield Express является однодокументным приложением и позволяет одновременно работать только с одним проектом.

Имя проекта (в примере — Personnel) добавляется в заголовок окна. В левой части окна находятся 6 групп страниц с параметрами и командами, предназначенными для задания характеристик дистрибутива, его создания и тестирования. Выбор группы приводит к отображению в правой части окна информации, которая поясняет назначение и особенности этой группы. На приведенном рисунке это группа **Organize Your Setup** (Организация инсталляции). При выборе страницы в левой части окна на панели справа выводится список названий и значений параметров, расположенных на этой странице. Так, на рис. 38.11 (здесь и далее приводится центральная часть окна программы) показаны параметры, размещенные на странице **General Information**.

Большинство действий при работе с параметрами выполняется с помощью команд контекстного меню объектов. После установки параметров слева от названия страницы появляется красная галочка. Для создания дистрибутива приложения не обязательно устанавливать значения всех параметров. Рассмотрим подробнее основные параметры дистрибутива.

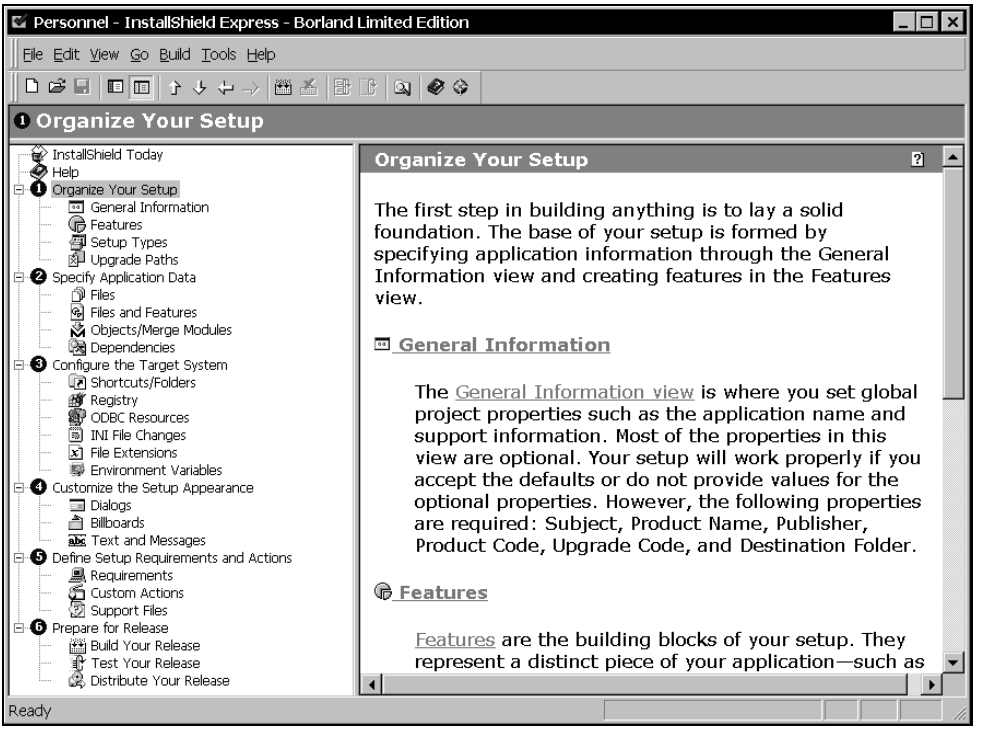


Рис. 38.10. Окно программы InstallShield Express

Product Properties	
Product Name	Application25
Product Version	1.00.0000
Product code	{51CE973F-3514-4321-8019-0F1DE3E2EC62}
Upgrade code	{5D703197-3D31-4677-B155-9BB787696417}
INSTALLDIR	[ProgramFilesFolder]Your Company Name\Your Product Name
DATABASEDIR	[INSTALLDIR]Database
Default Font	Tahoma:8
Summary Information Stream	
Author	Khomonenko, Gofman
Authoring Comments	Contact: Your local administrator
Subject	
Keywords	Installer,MSI,Database
Schema	200

Рис. 38.11. Параметры страницы **General Information**

Организация процесса инсталляции

Страница **Organize Your Setup** позволяет определить информацию об устанавливаемом приложении и вид главного окна инсталляционной программы. В эту группу входят следующие страницы:

- **General Information** (Общая информация);
- **Features** (Компоненты);

- **Setup Types** (Типы инсталляции);
- **Upgrade Paths** (Модернизация путей).

На странице **General Information** указываются основные параметры инсталляции приложения (продукта), данные о компании и о приложении. Отметим следующие параметры:

- `Author` (авторы);
- `Product Name` (название продукта);
- `Display Icon` (значок приложения);
- `Product Version` (версия);
- `INSTALLDIR` (каталог инсталляции);
- `Publisher/Product URL` (интернет-адрес производителя и продукта);
- `Publisher` (производитель);
- `Readme` (информационный файл);
- `Support Contact` (адрес организации, сопровождающей приложение);
- `Support Phone number` (телефон организации, сопровождающей приложение);
- `Product code` (код приложения);
- `Product Upgrade Code` (код для обновлений приложения);
- `DATABASEDIR` (расположение базы данных).

Параметр `Product Name` определяет имя, под которым приложение обозначается в Windows и отображается, например, при инсталляции с дистрибутива или при удалении с помощью компонента **Add/Remove Programs** (Установка и удаление программ) в Панели управления.

Параметр `INSTALLDIR` задает каталог, в котором размещаются файлы инсталлируемого приложения. Для задания этого каталога можно использовать параметры (имя параметра в квадратных скобках), конкретные значения (например, `D:\BookStore`), а также комбинацию параметров и значений (например, `[ProgramFilesFolder]\abc`). По принятым в Windows соглашениям приложение должно располагаться в каталоге Program Files, поэтому по умолчанию для `INSTALLDIR` предлагается значение `[ProgramFilesFolder]\[Company Name]\[Product Name]`. При инсталляции приложения вместо имени параметра подставляется соответствующее значение, например, название организации.

На странице **Features** можно определить компоненты (составные части) дистрибутива, в которые объединяются отдельные группы файлов и которые предоставляют те или иные возможности приложения. (Отметим, что эти компоненты не имеют никакого отношения к компонентам Delphi.) Объединение групп файлов в компоненты позволяет организовать различные режимы инсталляции приложения (например, минимальная или типовая).

Компоненты можно создавать, переименовывать, удалять, а также изменять порядок их следования. Эти действия выполняются с помощью соответствующих команд контекстного меню компонента, а при его создании — контекстного меню параметра

`Features`. Например, на рис. 38.12 к проекту добавлены два компонента с именами по умолчанию.

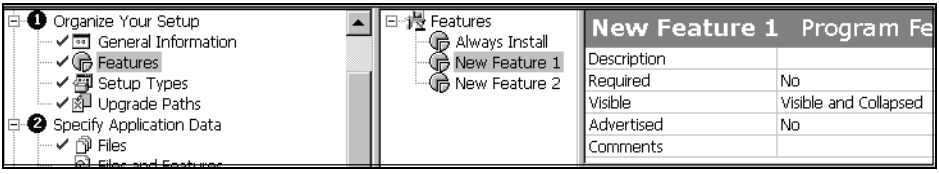


Рис. 38.12. Управление компонентами дистрибутива

Отметим, что на странице **Features** определяются только сами компоненты, а составляющие их файлы задаются на страницах **Files** и **Files and Features**.

По умолчанию у проекта есть один компонент **Always Install** (Инсталлировать всегда), который содержит файлы, устанавливаемые в любом случае.

На странице **Setup Types** можно распределить компоненты по следующим типам (вариантам) инсталляции:

- **Typical** (Типовая);
- **Minimal** (Минимальная);
- **Custom** (По усмотрению пользователя (выборочная)).

Названия типов выводятся в списке **Setup Types** (рис. 38.13). Слева от названия находится флажок, управляющий доступностью варианта при инсталляции приложения. По умолчанию флажки установлены, и при инсталляции приложения доступны все три типа. Чтобы исключить какой-либо вариант, нужно снять соответствующий флажок.

При выборе в списке **Setup Types** типа инсталляции в списке справа отображаются названия всех компонентов, включением которых в инсталляцию также можно управлять с помощью флажков. Так, в примере, показанном на рис. 38.13, при минимальной инсталляции приложения не будет установлен компонент **New Feature 1**.

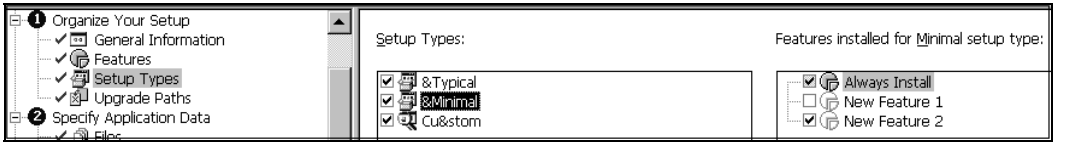


Рис. 38.13. Типы инсталляции приложения

Если компоненты не определены, то возможна только типовая инсталляция приложения.

Страница **Upgrade Path** используется в случае, когда необходимо модифицировать установленное приложение, не удаляя его предыдущую версию.

Общие установки

Группа **Specify Application Data** (Определить данные приложения) служит для указания файлов, входящих в состав приложения.

В эту группу входят следующие страницы:

- **Files** (Файлы);
- **Files and Features** (Файлы и компоненты);
- **Objects/Merge Modules** (Объекты/добавляемые модули);
- **Dependencies** (Зависимости).

Страница **Files** позволяет выбрать файлы, включаемые в дистрибутив приложения. Разработчик должен указать файлы и их каталоги на своем компьютере, а также размещение этих файлов на компьютере, где будет устанавливаться приложение (указываемое размещение может не совпадать с размещением на компьютере разработчика). Выбор файлов не отличается от действий в среде Проводника Windows. После указания каталога в дереве **Source computer's folders** исходный файл выбирается в списке **Source computer's files** и перетаскивается мышью в каталог инсталляции, отображаемый в списке **Destination computer's folders**. Список **Destination computer's files** отображает файлы выбранного каталога инсталляции. На рис. 38.14 главный каталог инсталляции приложения, определяемый как `INSTALLDIR`, содержит файлы FileView.exe и files.hlp, которые при инсталляции приложения будут размещены именно в этом каталоге.

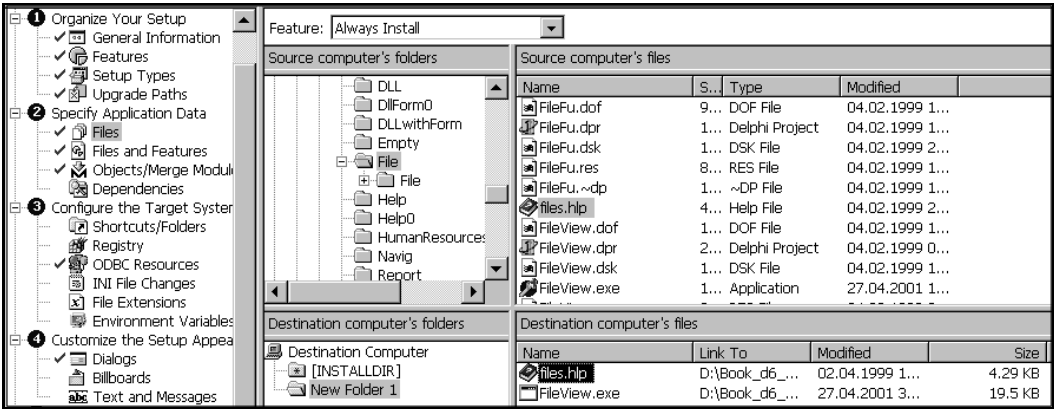


Рис. 38.14. Выбор файлов, включаемых в дистрибутив приложения

Разработчик может отобразить или скрыть ряд системных каталогов, выполнив команду **Show Predefined Folder** (Показать предопределенную папку) контекстного меню элемента **Destination Computer** (Компьютер размещения) и включив или выключив флажок для соответствующего каталога, например, MyPicturesFolder. Кроме предопределенных каталогов, можно создавать (командой **Add** контекстного меню) и использовать свои каталоги (на рис. 38.14 таким каталогом является **New Folder 1**). С помощью команд **Rename** и **Delete** контекстного меню можно соответственно переименовать и удалить свой каталог.

Размещаемые файлы можно удалять и перемещать между папками стандартным способом (например, с помощью Проводника).

Список **Feature** в верхней части окна определяет компонент, к которому относятся каталоги и выбранные для них файлы, по умолчанию это компонент **Always Install**.

На странице **Files and Features** можно просматривать списки файлов, принадлежащих тому или иному компоненту, а также удалять и перемещать файлы между компонентами.

Страница **Objects/Merge Modules** позволяет указать системные модули, необходимые для работы приложения, например Access 2000 или процессор баз данных BDE. Названия доступных модулей выводятся в списке **InstallShield Objects/Merge Modules**, нужный модуль добавляется к проекту установкой флажка слева от названия модуля (рис. 38.15).

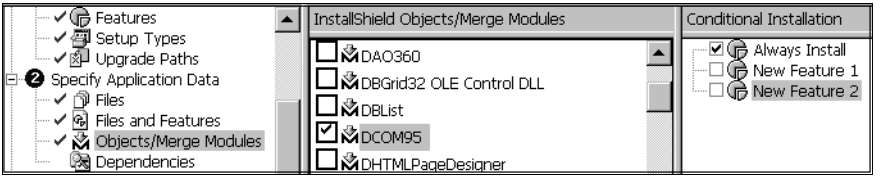


Рис. 38.15. Выбор системных модулей, необходимых для работы приложения

Если модуль нуждается в настройке, то при его выборе появляются дополнительные окна, позволяющие задать требуемые параметры. Так, для процессора баз данных BDE необходимо определить псевдонимы баз данных.

Флажки в списке **Conditional Installation** определяют компонент, вместе с которым происходит инсталляция модуля. По умолчанию указанный модуль устанавливается при любом типе инсталляции.

Если на компьютере пользователя выбранный модуль уже установлен, то при инсталляции приложения будет выполнено его обновление.

Настройка компьютера

Группа **Configure the Target System** (Конфигурировать целевую систему) служит для настройки компьютера, на котором устанавливаются приложения.

В эту группу входят следующие страницы:

- **Shortcuts/Folders** (Значки для быстрого доступа и папки);
- **Registry** (Реестр);
- **ODBC Resources** (Ресурсы ODBC);
- **Ini Files Changes** (Изменения в инициализационных (конфигурационных) файлах);
- **File Extensions** (Расширения файлов);
- **Environment variables** (Переменные окружения).

На странице **Shortcuts/Folders** можно определить значки для открытия файлов (запуска программ), а также значки и папки, которые будут добавлены к меню **Start** (Пуск) Windows и рабочему столу при инсталляции приложения. На рис. 38.16 показано создание на рабочем столе папки **New Folder 1**.

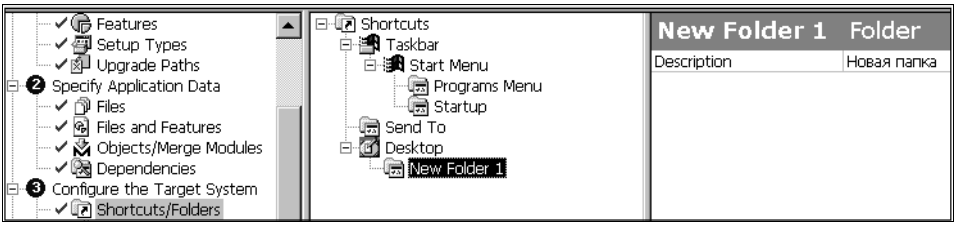


Рис. 38.16. Создание значков и папок в меню и на рабочем столе

Страница **Registry** позволяет внести изменения в системный реестр Windows, например, отредактировать значение ключа.

Замечание

Действия, связанные с изменением системного реестра, являются потенциально опасными и могут не только нарушить работу приложения, но и привести к разрушению всей системы. Поэтому подобные действия рекомендуется выполнять только хорошо подготовленным разработчикам.

Если распространяемое приложение использует технологию ODBC, то соответствующие ресурсы можно включить в состав дистрибутива на странице **ODBC Resources**. Нужный ресурс выбирается в иерархическом списке **ODBC Resources**, а в списке **Features** определяется компонент, в состав которого включается выбранный ресурс. Например, на рис. 38.17 в состав дистрибутива включается источник данных Birthdays, связанный с базой данных Access, которая находится в файле D:\myprojects\BirthDay \BirthDay.mdb.

По сравнению с изменением системного реестра более безопасным способом настройки параметров работы приложения является использование инициализационных (конфигурационных) файлов. Управление инициализационными файлами, входящими в состав дистрибутива, осуществляется на странице **INI File Changes**. Разработчик может добавить новый инициализационный файл, его раздел и новый параметр (рис. 38.18), а также переименовать и удалить их. Отдельные параметры можно перемещать между разделами и файлами, а разделы — между файлами. Эти действия выполняются командами контекстных меню соответствующих объектов.

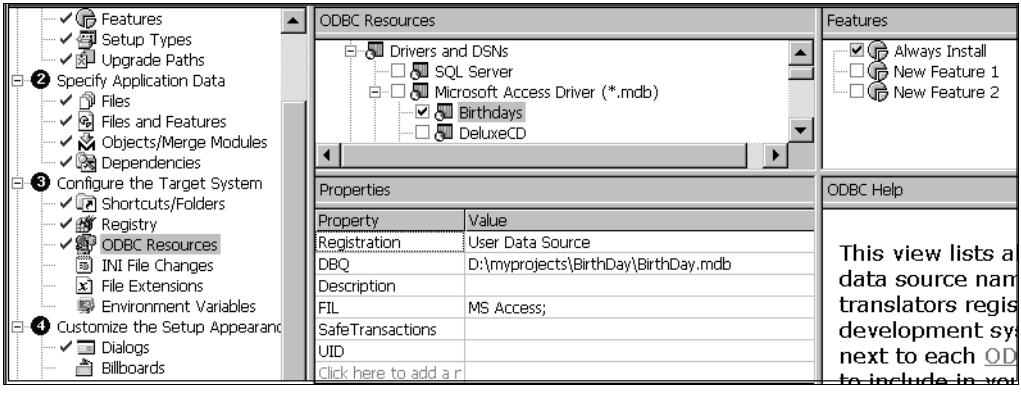


Рис. 38.17. Выбор ресурсов ODBC для включения в дистрибутив приложения

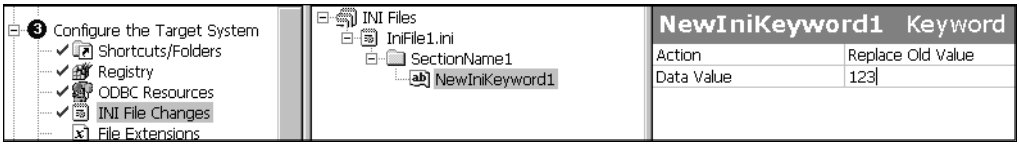


Рис. 38.18. Управление инициализационными файлами

На странице **File Extensions** задается ассоциированная связь между каким-либо расширением имени файла и приложением, автоматически запускаемым Windows для открытия файлов с таким расширением.

Страница **Environment variables** позволяет управлять переменными окружения Windows, определяя новые либо удаляя и изменяя существующие переменные.

Задание интерфейса процесса инсталляции

Для задания интерфейса процесса инсталляции используются возможности группы **Customize the Setup Appearance** (Настроить интерфейс установки). В эту группу входят следующие страницы:

- **Dialogs** (Диалоги);
- **Billboards** ("Доски объявлений");
- **Text and Messages** (Текст и сообщения).

На странице **Dialogs** определяется состав и вид диалоговых окон, с помощью которых программа инсталляции взаимодействует с пользователем. Для задания интерфейса инсталляционной программы можно использовать следующие диалоговые окна (рис. 38.19):

- **Splash Bitmap** (заставка с информацией об устанавливаемом приложении);
- **Install Welcome** (сообщение о начале процесса инсталляции);
- **License Agreement** (текст лицензионного соглашения, который содержится в заданном RTF-файле);
- **Readme** (сведения об устанавливаемом приложении);
- **Customer Information** (сведения о пользователе и серийном номере устанавливаемой копии приложения);
- **Destination Folder** (каталог инсталляции приложения);
- **Database Folder** (каталог для инсталляции базы данных);
- **Setup Type** (тип инсталляции);
- **Custom Setup** (выборочная (по усмотрению пользователя) инсталляция элементов);
- **Ready to Install** (готовность к началу инсталляции);
- **Setup Progress** (индикатор хода инсталляции);
- **Setup Complete Success** (сообщение об окончании инсталляции; при необходимости выводится запрос на перезагрузку компьютера).

Появлением диалога в процессе инсталляции управляет флажок, расположенный слева от названия диалогового окна.

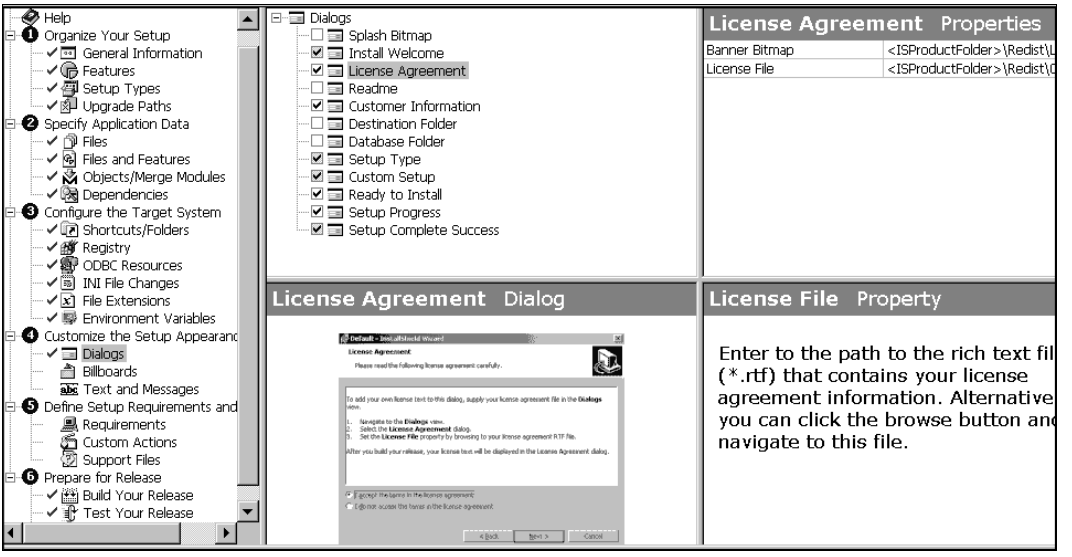


Рис. 38.19. Задание интерфейса программы инсталляции

При выборе в списке диалога в нижней панели выводится его вид, а в правой панели — список параметров. Диалоги являются настраиваемыми, т. е. позволяют изменять свои параметры. Наиболее общим параметром является изображение, определяющее вид диалога. Для большинства диалогов его можно изменять, нажав кнопку и выбрав нужный растровый файл (расширение bmp). Рисунок должен иметь размеры 497×55 пикселов (точек).

Состав остальных параметров зависит от выбранного диалога. Отметим параметр **License File**, задающий лицензионный файл для диалога **License Agreement**, в качестве которого можно выбрать файл типа RTF, содержащий текст лицензионного соглашения. По умолчанию используется файл Eula.rtf с текстом, поясняющим, как разместить на его месте текст разработчика. Параметр **Readme File** для диалога **Readme** указывает файл типа RTF, обычно содержащий краткую справочную информацию о приложении и особенностях его инсталляции.

Если в процессе инсталляции требуется обеспечить ввод и проверку серийного номера приложения, то используются параметры диалога **Customer Information**. Параметр **Serial Number Template** (Шаблон серийного номера) позволяет задать вид запроса для ввода серийного номера и допустимые символы. Шаблон представляет собой последовательность алфавитно-цифровых символов, при этом символы `#` и `?` приводят к отображению в диалоговом окне поля ввода, а остальные символы выводятся в качестве надписей. Символ `#` позволяет вводить в поле только цифры, в то время как символ `?` не ограничивает набор допустимых для ввода символов.

Например, если серийный номер имеет значение `123 ABCD17`, то шаблон может иметь вид `Серийный номер — ###-??????`.

Параметр **Serial Number Validation DLL** указывает динамическую библиотеку (DLL), которая выполняет проверку правильности введенного серийного номера. По умолчанию значения обоих параметров не заданы, и пользователю не выдается запрос на ввод серийного номера устанавливаемого приложения.

На странице **Billboards** можно определить окна, которые поочередно отображаются на экране в процессе установки и обычно содержат рекламную информацию о компании-производителе, устанавливаемом и других продуктах.

Страница **Text and Messages** служит для централизованного хранения текста, отображаемого в процессе установки приложения.

Определение дополнительной функциональности

Для добавления к программе инсталляции функциональности, не поддерживаемой службой инсталляции Windows (Windows Installer Service), используется группа **Define Setup Requirements and Actions** (Определить требования и действия по установке). В эту группу входят следующие страницы:

- **Requirements** (Требования);
- **Custom Actions** (Действия индивидуального пользователя);
- **Support Files** (Файлы поддержки).

Создание дистрибутива

После определения параметров дистрибутивного варианта приложения нужно создать сам дистрибутив. Для этого используется группа **Prepare for Release** (Подготовка для распространения), на которой расположены страницы:

- **Build Your Release** (Создание дистрибутива);
- **Test Your Release** (Проверка дистрибутива);
- **Distribute Your Release** (Распространение дистрибутива).

Создание дистрибутива заключается в подготовке файлов, которые затем должны быть скопированы на диски. Предварительно нужно выбрать устройство, на котором будут размещены файлы дистрибутива:

- CD-ROM;
- DVD;
- другое устройство.

Эти действия выполняются на странице **Build Your Release** (рис. 38.20), на которой также можно задать или изменить характеристики выбранного устройства, например, емкость CD.

После указания устройства процесс создания дистрибутива начинается с выполнения команды **Build | Build <*Устройство*>** или нажатия клавиши <F7>. В процессе по-

строения дистрибутива в нижней части окна программы InstallShield Express выводятся результаты. Файлы созданного дистрибутива записываются в каталог, имя которого совпадает с именем файла проекта. Кроме того, в этот каталог записываются вспомогательные файлы с информацией о созданном дистрибутиве: текстовые файлы помещаются в подкаталог LogFiles, а HTM-файлы — в подкаталог Reports.

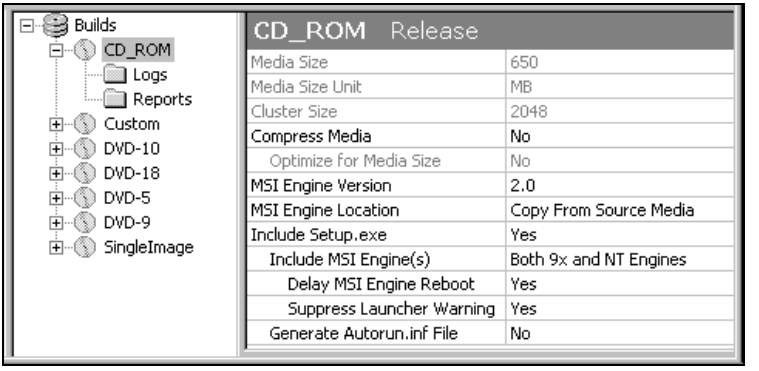


Рис. 38.20. Выбор устройства для размещения файлов дистрибутива

Перед копированием дистрибутива на выбранное устройство желательно выполнить тестирование созданного (на жестком диске) дистрибутива. Для этого задается команда **Build | Test <*Устройство*>**, в результате чего процесс инсталляции приложения с подготовленного дистрибутива выполняется так же, как он происходил бы при его реальной инсталляции на компьютере пользователя. Таким образом, на компьютере разработчика после тестовой инсталляции оказываются два варианта приложения: исходный

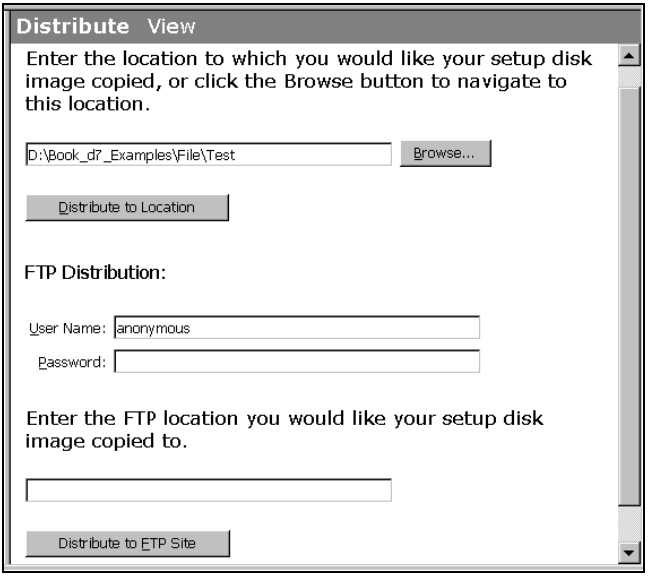


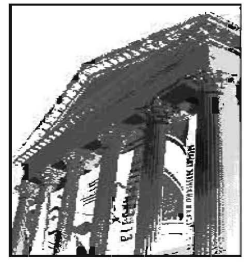
Рис. 38.21. Копирование файлов дистрибутива на внешний носитель

и новый (инсталлированный), который должен работать так же, как на компьютере пользователя.

Последним шагом является копирование созданного дистрибутива на носитель устройства, что выполняется на странице **Distribute Your Release** (рис. 38.21).

Для переноса файлов необходимо указать целевое (конечное) размещение дистрибутива, что удобно делать в окне, открываемом нажатием кнопки **Browse**. Само копирование файлов выполняется нажатием кнопки **Distribute to Location**.

С помощью протокола FTP дистрибутив также можно передать по сети, указав имя получателя (**User Name**), пароль (**Password**) и его FTP-адрес и нажав кнопку **Distribute to FTP Site**.



ГЛАВА 39

Библиотеки, пакеты и компоненты

При своем выполнении приложения Windows могут использовать средства, которые находятся в динамически подключаемых библиотеках (Dynamic Link Library, DLL). В Delphi библиотеки могут объединяться в пакеты времени разработки или выполнения.

В состав Delphi входит более ста компонентов; другие фирмы также предлагают широкий спектр компонентов. С их помощью можно создавать приложения для решения многих задач. При необходимости программист может разработать и собственные компоненты, например, когда имеющиеся компоненты не совсем подходят для решения поставленной задачи или нужный компонент просто отсутствует.

В этой главе рассматриваются использование библиотек DLL и пакетов библиотек, а также создание пользовательских компонентов.

Использование библиотек DLL

Динамически подключаемая библиотека DLL (далее библиотека, или DLL) не является исполняемой программой, но содержащийся в ней код может быть вызван из приложения или из другой DLL. Кроме кода, в библиотеке могут находиться ресурсы. Программист может использовать готовые или создавать собственные библиотеки.

Использование библиотек предоставляет ряд преимуществ.

- Экономия оперативной памяти — если несколько одновременно выполняемых приложений используют общую DLL, то она будет загружена в оперативную память только один раз.
- Удобное повторное использование кода — если какие-нибудь средства, например классы, управляющие устройствами для воспроизведения CD и DVD, должны использоваться различными приложениями, то их удобно поместить в библиотеку.
- Секционирование кода — большое приложение можно разделить на несколько секций (частей), одна из которых будет собственно приложением, а другие оформлены как библиотеки.

Секционирование позволяет:

- одновременно работать над проектом нескольким разработчикам, каждый из которых создает и отлаживает свою часть проекта;
- сэкономить оперативную память, если при выполнении приложения загружать требуемые библиотеки только по мере надобности — при непосредственном обращении к их коду. Например, в случае, когда при работе с приложением пользователь не всегда использует форму для работы со справочником, может оказаться целесообразным разместить форму в библиотеке. Эту библиотеку можно загружать только при обращении к содержащейся в ней форме, а по окончании работы с формой снова удалять из оперативной памяти, в этом случае приложение не будет занимать лишние ресурсы;
- облегчить модификацию проекта. Например, если в новой версии проекта изменились только несколько библиотек, то пользователю достаточно получить от разработчика измененные библиотеки, не переустанавливая все приложение.

♦ Обмен кодом между различными средами программирования — библиотека, созданная в одной среде программирования, например, в Delphi или C++, может быть применена в Visual Basic, Paradox, Excel и др. Возможен и обратный обмен — использование в Delphi библиотек, написанных на C++.

Исполняемая программа (exe-файл) и библиотека (dll-файл) имеют много общего, например похожую структуру. Принципиальное отличие состоит в том, что код библиотеки не будет самостоятельно выполнен, даже если он загружен в оперативную память. Этот код представляет собой лишь некоторый набор процедур и функций, которые могут вызываться другими программами или библиотеками.

Использование библиотеки динамической компоновки делится на два этапа:

♦ создание библиотеки;

♦ вызов библиотеки.

Отметим, что для одновременной отладки проектов библиотеки и вызывающего приложения удобно использовать Менеджер проектов, который позволяет удобно переключаться между проектами.

Библиотека и модуль

С точки зрения разделения кода библиотека похожа на обычный программный модуль (pas-файл). В библиотеке, как и в модуле, удобно размещать общий код, а также секционировать большой проект.

Однако принципиальное отличие заключается в том, что код модуля подключается к проекту *статически* — на этапе компиляции и компоновки проекта. При этом выполняется проверка кода модуля на правильность, а в исполняемый файл (exe) приложения попадает весь *используемый* код модуля, например, процедуры и функции. Отметим, что после компиляции модуля компоновщик включает в исполняемый файл именно используемый, а не весь код модуля.

Библиотека же подключается к приложению *динамически* — на этапе его выполнения. При компиляции проекта, использующего библиотеку, компоновщик только настраи-

вает в исполняемом файле специальные таблицы, занося туда информацию о внешних (экспортируемых из библиотек) процедурах и функциях. Сами же библиотеки загружаются после запуска вызывающего их приложения.

При использовании библиотечных функций компилятор не может проверить их код, а также наличие самой библиотеки. В связи с этим при возникновении каких-либо проблем с библиотекой или с вызванной из нее функцией ошибка появится уже при выполнении приложения.

Создание библиотеки

Создание библиотеки чаще всего начинается выбором объекта **DLL Wizard** на странице **New** в Хранилище объектов. Напомним, что приложение-сервер Web-службы также создается как библиотека. Первоначально проект библиотеки состоит из одного файла, который имеет вид:

```
library Project2;
{ Important note about DLL memory management: ShareMem must be the first unit in
your library's USES clause AND your project's (select Project-View Source) USES
clause if your DLL exports any procedures or functions that pass strings as
parameters or function results. This applies to all strings passed to and from
your DLL--even those that are nested in records and classes. ShareMem is the
interface unit to the BORLNDMM.DLL shared memory manager, which must be deployed
along with your DLL. To avoid using BORLNDMM.DLL, pass string information using
PChar or ShortString parameters. }
uses
SysUtils,
Classes;
{$R *.res}
begin
end.
```

Этот проект является заготовкой, куда необходимо добавить код, для содержания которого создана эта библиотека.

Ключевое слово `library` указывает, что это динамическая библиотека, а не исполняемый файл, и что при компиляции проекта будет создан файл с расширением dll, а не exe.

Код, располагаемый между операторными скобками `begin` и `end`, служит для инициализации библиотеки и автоматически выполняется при ее загрузке. При инициализации библиотеки можно проверить какие-либо условия, задать начальные значения переменным и т. п.

Достаточно длинный комментарий в проекте библиотеки предупреждает, что если функции библиотеки передают строки (в качестве параметров или результата), то в раздел `uses` самой библиотеки и вызывающих ее программ и других библиотек должен быть включен модуль `ShareMem`. Причем этот модуль должен быть первым в списке модулей. Кроме того, приложению и библиотеке при передаче строк требуется файл BorlndMM.dll.

Дальнейшее создание библиотеки заключается:

- в добавлении к проекту функций;
- в экспортировании их из библиотеки.

Описание функций располагают между секциями подключения модулей и инициализации. Функции кодируются обычным способом.

Экспортирование функций заключается в объявлении функции как внешней. С этой целью функция указывается в секции `exports`, которая имеет следующий вид:

```
exports
    <Имя функции>   [name <Внешнее имя функции>],
     ...
    <Имя функции>   [name <Внешнее имя функции>];
```

`Имя функции` — это имя, заданное при описании функции в проекте библиотеки. `Внешнее имя функции` представляет собой имя, под которым функция экспортируется и может быть вызвана извне DLL. Указывать внешнее имя не обязательно, в этом случае функция экспортируется под своим именем.

Кроме экспортируемых функций, библиотека может включать в себя внутренние (неэкспортируемые) функции, а также глобальные переменные.

В листинге 39.1 приведен пример простой библиотеки.

Листинг 39.1. Пример простой библиотеки

```
library SimpleDLL;
// При передаче строк не забудьте включить
// в этот раздел модуль ShareMem
uses ShareMem, SysUtils, Classes;
{$R *.res}
// Глобальная для DLL переменная
var GlobalVariable: real;

// Сложение целых чисел
function AddIntegers(a, b :integer): integer;
begin
  Result := a + b;
end;

// Сложение вещественных чисел
function AddReals(c, d :real): real;
begin
  Result := c + d;
end;

// Сложение строк
function AddStrings(e, f :string): string;
begin
  Result := e + f;
end;

// Внутренняя (неэкспортируемая) функция
function DoSomething(): integer;
begin
  Result := 1;
end;
```

```
// Экспортирование функций
exports
  AddIntegers,
  AddReals,
  AddStrings  name 'AddStringFromDLL';
begin
end.
```

Приведенная библиотека включает три экспортируемых функции `AddIntegers`, `AddReals` и `AddStrings` для сложения соответственно целых, вещественных чисел и строк. Функции сложения чисел экспортируются под своими именами, а для функции сложения строк задано внешнее имя `AddStringFromDLL`, отличное от ее имени в библиотеке.

В связи с тем что функция передает строковые данные, первым в списке раздела `uses` указан модуль `ShareMem`.

Кроме экспортируемых функций, приведенная библиотека содержит неэкспортируемую функцию `DoSomething` и глобальную для библиотеки переменную `GlobalVariable`, которые приведены в качестве примера и в этой библиотеке не используются.

Вызов библиотеки

Под *вызовом библиотеки* понимается использование заложенных в нее кода и ресурсов. Рассмотрим вызов экспортируемых библиотекой функций.

При вызове библиотеки программист не имеет возможности указывать ее размещение, поэтому библиотека должна быть расположена в каталоге, в котором Windows обнаружит ее автоматически. Такими каталогами являются, например, корневой каталог Windows, его подкаталоги SYSTEM и SYSTEM32, а также каталог, из которого запущено приложение. Если при выполнении приложения необходимая библиотека не найдена, то выдается соответствующее сообщение (рис. 39.1).

Рис. 39.1. Сообщение об ошибке поиска библиотеки

Различают два способа вызова:

- статический;
- динамический.

Отметим, что ранее термины "статический" и "динамический" использовались в совершенно другом смысле. В данном случае эти термины не совсем корректны, т. к. при обоих способах вызова библиотека загружается в оперативную память именно во время выполнения приложения, т. е. динамически. Эти способы вызова также называют еще *неявным* и *явным*, что не приводит к некорректности, связанной с термином "статический", однако тоже не совсем верно в том смысле, что в обоих вариантах вызова имена библиотек и функций указываются в общем-то явно, хотя и разными способами.

При *статическом вызове* библиотека загружается в оперативную память вместе с использующими ее приложениями и другими библиотеками и выгружается после прекращения работы последнего использующего ее процесса. Достоинством статического вызова является простота реализации в программе. Однако при таком вызове программист не может управлять процессом загрузки библиотеки и, кроме того, библиотека находится в оперативной памяти все время работы приложения, даже если она не будет использована (ни одна из ее функций не будет вызвана).

Для статического доступа к библиотечной функции необходимо объявить ее как внешнюю (импортируемую). Это объявление располагается в разделе реализации модуля и имеет следующий вид:

```
<Заголовок функции>  external  <Имя библиотеки>;
```

`Заголовок функции` должен совпадать с заголовком функции из библиотеки, но в качестве имени функции указывается ее внешнее имя из раздела `exports`. `Имя библиотеки` представляет собой строку с именем файла, также содержащим расширение dll.

После объявления функции как внешней ее можно использовать как обычную внутреннюю функцию модуля.

В качестве примера рассмотрим приложение, из которого вызываются функции ранее созданной библиотеки SimpleDLL. В форме приложения (рис. 39.2) расположены три кнопки, заголовки которых (**Add Integers**, **Add Reals** и **Add Strings**) соответствуют именам функций, вызываемых из библиотеки. Нажатие каждой кнопки приводит к сложению двух значений из компонентов `Edit`, расположенных слева от кнопки, и помещению результата в компонент справа от кнопки. Сложение данных выполняет соответствующая функция (`AddIntegers`, `AddReals` или `AddStrings`), импортируемая из библиотеки SimpleDLL.

В листинге 39.2 приведен код модуля формы.

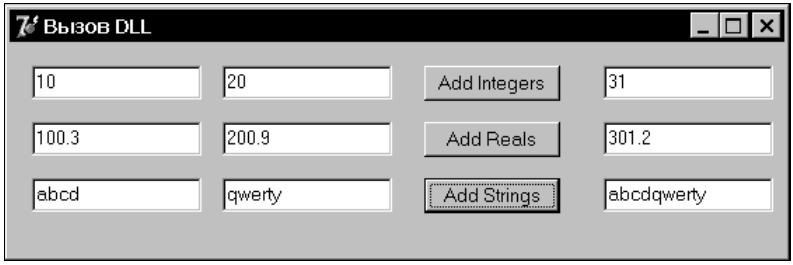


Рис. 39.2. Форма приложения для вызова библиотечных функций

Листинг 39.2. Пример приложения для вызова библиотечных функций

```
unit uCallDLL;
interface
// При передаче строк не забудьте включить
// в этот раздел модуль ShareMem
uses  ShareMem,
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    btnAddIntegers: TButton;
       btnAddReals: TButton;
     btnAddStrings: TButton;
       edtInteger1: TEdit;
          edtReal1: TEdit;
        edtString1: TEdit;
       edtInteger2: TEdit;
          edtReal2: TEdit;
        edtString2: TEdit;
  edtIntegerResult: TEdit;
     edtRealResult: TEdit;
   edtStringResult: TEdit;
    procedure btnAddIntegersClick(Sender: TObject);
    procedure btnAddRealsClick(Sender: TObject);
    procedure btnAddStringsClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
// Объявление функций как внешних (импортируемых).
// В качестве имен функций указываются
// их внешние имена из раздела exports библиотеки.
// Для имени файла библиотеки не забудьте указать расширение dll
function AddIntegers(a, b :integer): integer;
         external 'SimpleDLL.dll';
function AddReals(c, d :real): real;
         external 'SimpleDLL.dll';
function AddStringFromDLL(e, f :string): string;
         external 'SimpleDLL.dll';

// Сложение целых чисел
procedure TForm1.btnAddIntegersClick(Sender: TObject);
begin
  edtIntegerResult.Text := IntToStr(AddIntegers(
    StrToInt(edtInteger1.Text),
```

```
    StrToInt(edtInteger2.Text)));
end;

// Сложение вещественных чисел
procedure TForm1.btnAddRealsClick(Sender: TObject);
begin
  edtRealResult.Text := FloatToStr(AddReals(
    StrToFloat(edtReal1.Text),
    StrToFloat(edtReal2.Text)));
end;

// Сложение строк
procedure TForm1.btnAddStringsClick(Sender: TObject);
begin
  edtStringResult.Text := AddStringFromDLL(
    edtString1.Text,
    edtString2.Text);
end;
```

При объявлении внешней функции сложения строк указано ее внешнее имя `AddStringFromDLL`, т. к. имя `AddStrings` не экспортировано из библиотеки и, соответственно, недоступно извне.

В связи с тем что функция `AddStringFromDLL` передает строки, в раздел `uses` включен модуль `ShareMem`.

В приведенном примере при вводе в поля редактирования вещественных чисел в качестве десятичного разделителя использована точка. Следует иметь в виду, что вид разделителя зависит от настроек Windows.

При *динамическом вызове* библиотечных функций программист может управлять процессом загрузки библиотек, а также сэкономить оперативную память, загружая только нужные в данный момент библиотеки. Если какая-либо библиотека, например, содержащая форму для настройки параметров приложения, требуется относительно редко, то имеет смысл загружать ее только на время использования содержащихся в ней кода и ресурсов.

Динамический вызов требует от программиста намного больших усилий, чем в случае организации статического вызова. Кроме собственно вызова функции, программист должен обеспечить:

- загрузку библиотеки;
- получение адреса функции;
- выгрузку библиотеки.

Загрузку библиотеки выполняет API-функция `LoadLibrary(lpLibFileName: PChar): HMODULE`, которая загружает в оперативную память библиотеку с именем `lpLibFileName`. В качестве результата функция возвращает ссылку `HMODULE` на модуль библиотеки при успешной загрузке и `0` — при ошибке. Напомним, что программист не может указать расположение библиотеки, и она должна быть расположена в каталоге, где будет автоматически найдена Windows.

Выгружается библиотека API-функцией `FreeLibrary(hLibModule: HMODULE): Bool`, удаляющей из оперативной памяти библиотеку, модуль которой указывает параметр `HMODULE`. В случае успешной выгрузки функция возвращает `True` и `False` — в противном случае.

Получение адреса функции выполняет API-функция `GetProcAddress(hModule: HMODULE, lpProcName: LPCSTR): FARPROC` — она ищет в библиотеке, модуль которой задан параметром `hModule`, функцию по имени, указанному параметром `lpProcName`, и в качестве результата возвращает указатель на функцию. Если поиск оказался безрезультатным, то возвращается `Nil`.

С помощью полученного указателя можно вызвать библиотечную функцию. Перед вызовом указатель приводится к типу функции, что требует написания достаточно сложного кода.

В листинге 39.3 приведен код модуля формы, в которой выполняется вызов из библиотеки SimpleDLL функций сложения целых и вещественных чисел.

Листинг 39.3. Пример вызова функций сложения целых и вещественных чисел

```
unit uCallDLL2;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
// Типы внешних (из библиотек) функций
type TAddIntegers = function(p1, p2: integer): integer;
       TAddReals = function(p1, p2: real): real;
type
  TForm2 = class(TForm)
       edtInteger1: TEdit;
       edtInteger2: TEdit;
    btnAddIntegers: TButton;
  edtIntegerResult: TEdit;
          edtReal1: TEdit;
          edtReal2: TEdit;
       btnAddReals: TButton;
     edtRealResult: TEdit;
    procedure btnAddIntegersClick(Sender: TObject);
    procedure btnAddRealsClick(Sender: TObject);
  private
    // Ссылка на модуль DLL
    DLLHandle: THandle;
    // 1-й вариант переменной для приведения типа функции
    ExternalFunctionAddIntegers: TAddIntegers;
    // 2-й вариант переменных для приведения типа функции
    ExternalFunctionAddRealsPointer: TFarProc;
    ExternalFunctionAddReals:        TAddReals;
  public
    { Public declarations }
  end;
```

```
var
  Form2: TForm2;
implementation
{$R *.dfm}

procedure TForm2.btnAddIntegersClick(Sender: TObject);
var i1, i2, res: integer;
begin
  // Загрузка библиотеки
  DLLHandle := LoadLibrary('SimpleDll');
  if (DLLHandle = 0) then begin
    ShowMessage('Ошибка загрузки библиотеки SimpleDll!');
    exit;
  end;
  try
    // Получение указателя на функцию AddIntegers
    // и преобразование его к соответствующему типу
    @ExternalFunctionAddIntegers :=
      GetProcAddress(DLLHandle, PChar('AddIntegers'));
    // Преобразование данных к целочисленному типу
    try
      i1 := StrToInt(edtInteger1.Text);
      i2 := StrToInt(edtInteger2.Text);
    except
      ShowMessage('Ошибка данных!');
      exit;
    end;
    // Вызов функции AddIntegers
    res := ExternalFunctionAddIntegers(i1, i2);
    edtIntegerResult.Text := IntToStr(res);
    // Выгрузка библиотеки
  finally
    FreeLibrary(DLLHandle);
  end;
end;

procedure TForm2.btnAddRealsClick(Sender: TObject);
var r1, r2, res:  real;
begin
  // Загрузка библиотеки
  DLLHandle := LoadLibrary('SimpleDll');
  if (DLLHandle = 0) then begin
    ShowMessage('Ошибка загрузки библиотеки SimpleDll !');
    exit;
  end;
  try
    // Получение указателя на функцию AddReals
    // и преобразование его к соответствующему типу
    ExternalFunctionAddRealsPointer :=
                    GetProcAddress(DLLHandle, PChar('AddReals'));
```

```
    if (ExternalFunctionAddRealsPointer <> Nil)
      then ExternalFunctionAddReals :=
         FarProc(ExternalFunctionAddRealsPointer);
    // Преобразование данных к вещественному типу
    try
      r1 := StrToFloat(edtReal1.Text);
      r2 := StrToFloat(edtReal2.Text);
    except
      ShowMessage('Ошибка данных!');
      exit;
    end;
    // Вызов функции AddReals
    res := ExternalFunctionAddReals(r1, r2);
    edtRealResult.Text := FloatToStr(res);
    // Выгрузка библиотеки
  finally
    FreeLibrary(DLLHandle);
  end;
end;
end.
```

В приведенном примере библиотека динамически загружается при каждом вызове функции и выгружается сразу после выполнения функции. Для ссылки на библиотеку используется переменная `DLLHandle` типа `THandle`, который совпадает с типом `HMODULE`.

Для преобразования указателей на функции описаны типы `TAddIntegers` и `TAddReals`, которые должны быть равны типам соответствующих функций из библиотеки, т. е. иметь совпадающие порядок и тип параметров, а также тип возвращаемого результата. При этом имена параметров значения не имеют и могут различаться.

Преобразование указателя к типу функции выполнено для обеих функций по-разному. Так, для функции `AddIntegers` указатель, возвращаемый API-функцией `GetProcAddress`, непосредственно присваивается переменной `ExternalFunctionAddIntegers`, имеющей тип `TAddIntegers`. Для преобразования указателя на функцию `AddReals` к типу `TAddReals` (переменная `ExternalFunctionAddReals`) использована промежуточная переменная-указатель `ExternalFunctionAddRealsPointer` типа `TFarProc`.

Вызов функций осуществляется через переменные `ExternalFunctionAddIntegers` и `ExternalFunctionAddReals`. При вызове функций выполняется обработка исключений, которые могут возникнуть, например, в случае ошибки преобразования строки к числовому типу или при выполнении функции. Чтобы обеспечить выгрузку библиотеки даже при возникновении ошибки, функция `FreeLibrary` расположена в секции `finally`.

Отметим, что переменные включены в раздел `private` класса формы, а не в раздел реализации, как это делалось в предыдущих примерах.

Использование форм в библиотеках

Библиотека может содержать формы, при этом форма добавляется к библиотеке и конструируется обычным для проекта способом. Новым является то, что необходимо

включить в модуль формы экспортируемые функции (как минимум одну) для управления формой извне библиотеки.

В листинге 39.4 приводится код модуля, описывающий форму `FormInDLL`, которая входит в библиотеку `DLLwithForm`.

Листинг 39.4. Пример формы в составе библиотеки `DLLwithForm`

```
unit uFormInDLL;
interface
uses ShareMem,
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs;
type
  TFormInDLL = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  FormInDLL: TFormInDLL;
function ShowFormInDLL(Name: string): TModalResult;
implementation
{$R *.dfm}

function ShowFormInDLL(Name: string): TModalResult;
begin
  FormInDLL := TFormInDLL.Create(Application);
  FormInDLL.Caption := Name;
  Result := FormInDLL.ShowModal;
  FormInDLL.Free;
end;
end.
```

Для управления формой в состав модуля включена функция `ShowFormInDLL`, которая выполняет создание формы, отображение ее в модальном режиме и последующее удаление. Строковый параметр `Name` функции определяет заголовок формы, а в качестве результата возвращается модальный результат закрытия формы. Для обеспечения видимости функции `ShowFormInDLL` ее заголовок включен в раздел интерфейса модуля.

Чтобы обеспечить видимость функции, управляющей формой, имя функции указано в разделе `exports` файла проекта библиотеки, который приводится далее.

```
library DLLwithForm;
uses ShareMem,
  SysUtils,
  Classes,
  uFormInDLL in 'uFormInDLL.pas' {FormInDLL};
{$R *.res}
exports ShowFormInDLL;
begin
end.
```

Запуск формы, содержащейся в библиотеке, осуществляется вызовом экспортируемой функции, которая описана ранее. Для рассматриваемого примера это может быть выполнено следующим образом:

```
function ShowFormInDLL(name: string): TModalResult;
         external 'DLLwithForm.dll';
procedure TForm2.Button1Click(Sender: TObject);
begin
  Caption := IntToStr(ShowFormInDLL('Hello'));
end;
```

Приведенный код располагается в разделе реализации модуля приложения и выполняет статический вызов формы из библиотеки DLLwithForm.dll.

Отметим, что организовать взаимодействие между формами приложения и библиотеки сложнее, чем в случае, когда они находятся в составе общего приложения. Если необходимо передать в библиотечную форму указатель на приложение или его форму, из которых вызвана эта библиотечная форма, то это можно сделать, используя свойство `Handle` приложения или формы.

Особенности библиотек, предназначенных для различных сред разработки

Ранее мы рассмотрели библиотеки, создаваемые и используемые в среде Delphi. Библиотеки, разрабатываемые для использования в других средах разработки, например, C++ Builder, имеют ряд особенностей.

Так, в заголовках экспортируемых функций необходимо указывать директиву `stdcall`, предписывающую использовать стандартные соглашения вызова функции. По умолчанию Delphi использует директиву `register`, соответствующую быстрому типу вызова.

Для параметров и результатов функций следует применять базовые типы данных Windows, а не родные типы Delphi. В первую очередь это относится к строкам, которые должны быть преобразованы к типу `PChar`.

Системные библиотеки

Windows включает в свой состав разнообразные DLL, которые называют *системными* и которые предоставляют большое число различных средств. Для Windows центральными DLL являются 32-разрядные kernel32.dll, user32.dll и gdi32.dll и 16-разрядные kernel.exe, user.exe и gdi.exe, отсутствующие в Windows NT. Другие системные DLL расширяют возможности центральных DLL и предоставляют элементы управления, диалоги, шрифты, драйверы устройств и другие средства.

Разработчик достаточно широко использует средства системных DLL. Так, при выборе файла для открытия удобно использовать стандартный диалог `OpenDialog1`, который рассмотрен в *главе 4*, посвященной работе с формами. При вызове этого диалога методом `Execute` будет использован соответствующий диалог Windows, содержащийся в системной DLL. Реализация DLL зависит от версии Windows, поэтому при выполнении

приложения на различных компьютерах вид диалогов будет отличаться. Например, диалог, показанный на рис. 4.16, соответствует Windows NT.

Системные DLL также используются при вызове API-функций. Эти функции определены в модуле Windows.pas. В *главе 9*, посвященной организации приложений, для создания одноэкземплярного приложения использованы API-функции `FindWindow` и `SetForegroundWindow`. Далее приведены фрагменты модуля Windows.pas, относящиеся к этим функциям.

```
unit Windows;
...
Interface
...
function FindWindow(lpClassName, lpWindowName: PChar): HWND; stdcall;
function SetForegroundWindow(hWnd: HWND): BOOL; stdcall;
...
const
  advapi32 = 'advapi32.dll';
  kernel32 = 'kernel32.dll';
  mpr      = 'mpr.dll';
  {$EXTERNALSYM version}
  version  = 'version.dll';
  comctl32 = 'comctl32.dll';
  gdi32    = 'gdi32.dll';
  opengl32 = 'opengl32.dll';
  user32   = 'user32.dll';
  wintrust = 'wintrust.dll';
  msimg32  = 'msimg32.dll';
...
implementation
...
{ Externals from user32.dll }
function FindWindow; external user32 name 'FindWindowA';
function SetForegroundWindow; external user32 name 'SetForegroundWindow';
```

В интерфейсной части объявлены имена функций, состав параметров и возвращаемый результат, знать которые необходимо при вызове функций. В разделе реализации для функций вместо кода находится ссылка на внешнее описание в DLL. Здесь этой DLL является user32.dll, имя которой задано через константу `user32`, определенную в разделе интерфейса. Функция `FindWindow` вызывает из системной DLL функцию `FindWindowA`, указанную после директивы `name`. В связи с тем что функция `SetForegroundWindow` вызывает одноименную функцию системной DLL, для нее директива `name` является излишней.

Кроме user32.dll, в модуле Windows используются API-функции из других DLL, например, gdi32.dll и wintrust.dll.

Отметим, что директива `{$EXTERNALSYM version}` необходима для совместимости с C++ Builder и предназначена для блокирования записи имени `version` в заголовочном файле C++.

Использование пакетов

В Delphi библиотеки могут объединяться в так называемые *пакеты*. Есть два типа пакетов: пакеты времени разработки (design packages) и пакеты времени выполнения (runtime packages).

Пакеты времени разработки используются на этапе проектирования приложения и содержат код вызова объектов VCL, а также редакторов свойств. Функционирование этих пакетов заключается в вызове соответствующих пакетов времени выполнения.

К примеру, в интегрированной среде Delphi (IDE) для обеспечения использования Палитры компонентов используются следующие пакеты времени разработки:

- DCLSTD70 — страницы **Standard**, **Additional**, **System**, **Win32** и **Dialogs**;
- DCLDB70 — страницы **Data Access** и **Data Controls**;
- DCLRave70 — страница **Rave**.

Пакеты времени выполнения применяются на этапе выполнения приложения и содержат код VCL и модулей Delphi. В частности, в Delphi используются следующие пакеты времени выполнения:

- VCL — основные компоненты, системные функции, модули;
- VCLX — дополнительные компоненты;
- RTL — некомпонентные системные функции и элементы интерфейса Windows;
- VCLDB — компоненты баз данных;
- Rave50VCL — компоненты отчетов Rave 5.0.

Деление пакетов на два вида не является жестким, например, пакеты времени разработки можно использовать и как пакеты времени выполнения.

Применение пакетов позволяет:

- уменьшить объем приложения;
- предоставить нескольким приложениям, использующим общий пакет, единый интерфейс;
- облегчить изменение приложений.

Уменьшение объема приложения является главным достоинством применения пакетов и заключается в устранении избыточности кода. При создании приложения его исполняемый файл (exe) содержит не только функциональность, внесенную в него программистом, но и большое количество элементов, общих для многих приложений. По умолчанию пакеты не используются и исполняемый файл даже простейшего приложения имеет размер более 350 Кбайт. Однако этот файл является *автономным* и не требует для своего выполнения других файлов, в том числе библиотек и пакетов.

Для *уменьшения размера* исполняемого файла к приложению подключаются пакеты. Для этого на странице **Packages** окна параметров проекта нужно установить флажок **Build with runtime packages**. Подключаемые пакеты отображаются в списке, разработчик может управлять подключением пакетов с помощью флажков, расположенных слева от названий пакетов (рис. 39.3).

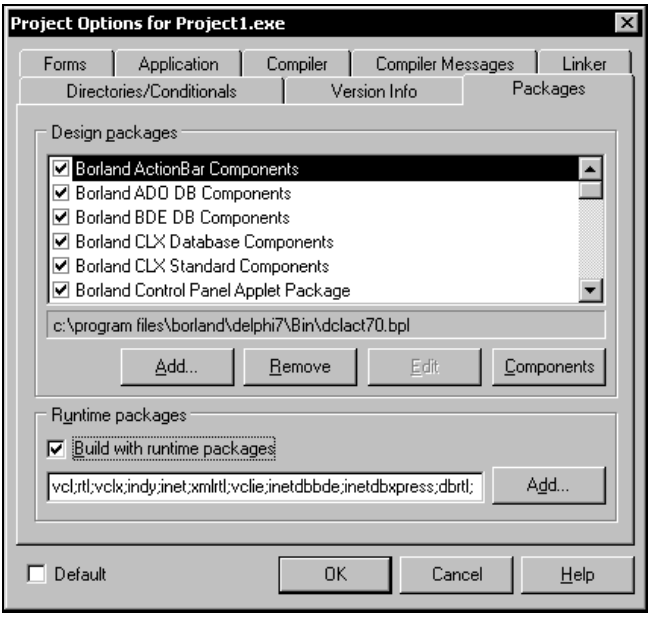


Рис. 39.3. Диалоговое окно подключения пакетов

В результате при компиляции в приложение войдет автоматически создаваемый для любого приложения код, а также собственно функциональный код приложения. За пределами приложения остается код, находящийся в пакетах, например, код для компонентов из состава библиотеки визуальных компонентов. При этом для простейшего приложения размер исполняемого файла уменьшается примерно до 15 Кбайт, т. е. более чем в 20 раз по сравнению с приложением, не использующим пакеты. Однако исполняемый файл теперь уже не является автономным и требует для своей работы наличия пакетов, использованных при создании приложения.

Пакеты имеют достаточно большой размер, поэтому применение пакетов имеет преимущества в случае, когда на одном компьютере находится несколько приложений, использующих эти пакеты.

При переносе приложения на другой компьютер вместе с ним должны быть перенесены и используемые пакеты. Файлы пакетов должны быть расположены в одном из тех каталогов, где Windows осуществляет поиск при выполнении приложения. Пакеты можно скопировать обычным способом или при помощи программы инсталляции InstallShield, которая, создавая дистрибутивный комплект, позволяет включить в него требуемые пакеты. При переносе последующих приложений они будут обращаться к пакетам, уже имеющимся на компьютере.

Замечание

В приложении нельзя указать расположение пакетов, поэтому они должны находиться или в системном каталоге, или в каталоге, который указан в списке каталогов для поиска.

Пакеты удобно применять при *отладке* приложений и в случае, когда распространяемое приложение должно быть создано без пакетов. В этом случае при применении па-

кетов исполняемые файлы отлаживаемых приложений занимают на диске относительно немного места. Перед переносом отлаженного приложения оно компилируется без учета пакетов и является автономным.

Создание компонентов

В составе Delphi поставляется более ста компонентов, большое число компонентов предлагается другими фирмами. С их помощью можно создавать приложения для решения широкого круга задач. При необходимости программист может разработать и собственные компоненты, например, в случаях, когда имеющиеся компоненты не совсем подходят для решения поставленной задачи или вообще отсутствует нужный компонент.

Создание компонента включает следующие этапы:

1. Создание шаблона класса — выбор компонента-предка для оформления на его основе описания класса.
2. Описание класса нового компонента:
   - задание новых свойств и методов компонента;
   - переопределение методов предка компонента.
3. Создание значка компонента.
4. Инсталляция компонента.

Рассмотрим этапы создания компонента более подробно на примере компонента-кнопки `BtnSound`, при нажатии которой, кроме выполнения обработчика события, издается звук.

Создание шаблона класса

Создание нового компонента начинается выбором объекта **Component** страницы **New** в Хранилище объектов или выполнением команды **Component | New Component**. При этом открывается диалоговое окно **New Component** (рис. 39.4).

В этом окне выполним следующие действия.

1. Выберем в комбинированном списке **Ancestor type** класс компонента-предка. В качестве предка обычно выбирается ближайший по функциональным возможностям класс. Так как создаваемая кнопка больше всего похожа на простую кнопку `Button`, выберем класс `TButton`.
2. Укажем в поле **Class Name** имя класса создаваемого компонента; по умолчанию предлагается имя класса-предка, к которому добавлена цифра 1. Лучше задать более осмысленное имя. В нашем случае введем `TBtnSound`, буква `T` означает, что имя относится к типу класса, сам компонент будет называться `BtnSound`.
3. Зададим в комбинированном списке **Palette Page** имя страницы в Палитре компонентов, на которую будет помещен новый компонент при его добавлении. Можно выбрать в списке любую из страниц или задать имя новой страницы, которая будет

автоматически создана при добавлении компонента. Оставим имя **Samples** страницы без изменений.

4. Зададим в поле **Unit file name** имя файла модуля с расширением pas, в котором сохраняется описание класса нового компонента. Имя файла является полным и включает в себя путь. По умолчанию новый модуль сохраняется в подкаталоге LIB главного каталога Delphi, а имя файла совпадает с названием компонента, в нашем случае `BtnSound`. Файл модуля следует размещать в каталоге, который может быть найден при работе с ним из среды Delphi.
5. В поле **Search path** перечисляются каталоги, в которых выполняется поиск файла модуля. Так как подкаталог LIB входит в этот список, оставим имя файла модуля без изменений.

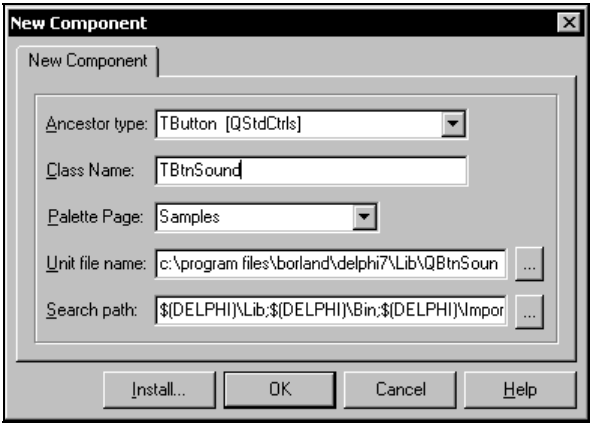


Рис. 39.4. Диалоговое окно создания компонента

После заполнения всех полей и нажатия кнопки **OK** создается модуль `BtnSound`, текст которого отображается в редакторе кода. Этот модуль содержит описание класса `TBtnSound`, созданного на основе класса `TButton`.

Модуль также содержит процедуру `Register`, которая будет выполнять регистрацию нового компонента `ButtonSound` на странице **Samples** Палитры компонентов. Первым параметром процедуры регистрации является название страницы, вторым — множество классов добавляемых компонентов (в нашем случае состоящее из одного класса `TBtnSound`).

Текст названных модуля и процедуры имеет следующий вид (листинг 39.5).

Листинг 39.5. Пример создания компонента

```
unit BtnSound;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
```

```
type
 TBtnSound = class(TButton)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TBtnSound]);
end;
end.
```

В дальнейшем к классу `TBtnSound` добавляется код, который отличает этот класс и компонент от его предка. Функциональность нового компонента определяется описанием класса и заключается в создании новых полей, свойств (в том числе событий) и методов, а также в переопределении методов класса-предка. Новые характеристики соответствуют новому назначению компонента, а переопределению подлежат методы, которые выполняются отличным от методов предка образом.

Создание свойств

Класс содержит поля, в которых хранится информация данного класса. По соглашению имена полей начинаются с буквы `F`. Обычно поля описываются в разделе частных (`Private`) описаний и не доступны для прямого программного обращения к ним. Программно доступ к значениям полей осуществляется через свойства, которые обычно имеют имена, определяющие их принадлежность к конкретному полю. Свойства, которые должны быть доступны программно, описываются в разделе опубликованных и общедоступных (`Published` и `Public`) описаний, в противном случае их помещают в защищенный раздел (`Protected`) описаний. Описание свойства имеет следующий формат:

```
property <Имя свойства > : <Тип свойства> <Спецификаторы>;
```

Основными спецификаторами свойства являются `read` и `write`, соответственно определяющие возможность *чтения* и *записи* значения свойства. При этом необходимо указать хотя бы один из этих спецификаторов. После спецификатора указывается имя поля или метода. Если значение свойства *непосредственно* связано со значением своего поля, то в спецификаторе указывается имя этого поля. Если используются оба спецификатора, то при непосредственной связи свойства и поля формат описания свойства имеет вид:

```
property <Имя свойства> : <Тип свойства> read <Имя поля> write <Имя поля>;
```

Если взаимосвязь между свойством и полем имеет более сложный характер, то после спецификатора указывается имя метода, который считывает или устанавливает значе-

ние свойства. Метод *чтения* является функцией без параметров, которая имеет тот же тип, что и тип читаемого свойства. По соглашению имя функции начинается со слова `Get`. Метод *записи* является процедурой с одним параметром того же типа, что и устанавливаемое свойство. Внутри методов можно реализовать достаточно сложные алгоритмы чтения и записи значения свойства, в том числе с учетом значений других свойств, полей или иных переменных. Если применяются оба спецификатора, то при использовании для доступа к значению свойства методов формат описания свойства имеет вид:

```
property <Имя свойства> : <Тип свойства> read <Имя метода> write <Имя метода>;
```

Спецификаторы `read` и `write` используются независимо друг от друга, и для любого из них можно указывать имя поля или имя метода. Если отсутствует спецификатор `read`, то значение свойства нельзя считывать, а если не указан спецификатор `write`, то свойство предназначено только для чтения (такой вариант встречается чаще).

Как отмечалось, для работы со свойством через Инспектор объектов для каждого изменяемого свойства автоматически подключается специальный редактор свойства. Ниже перечислены общие виды свойств.

- *Простые* — принадлежат целочисленному, символьному или строковому типу. В Инспекторе объектов значения этих свойств отображаются как обычный текст, поэтому простые свойства также называют *текстовыми*. Например, простыми свойствами являются свойства `Width` и `Caption`.
- *Перечислимые* — принадлежат перечислимому или логическому типу и позволяют выбрать одно значение из предлагаемого списка. Например, перечислимыми являются свойства `Align` и `Visible`.
- *Множественные* — принадлежат множественному типу и позволяют задать любую комбинацию значений из предлагаемого набора. Например, множественными свойствами являются `BorderIcons` и `Anchors`.
- *Объекты* — эти свойства, в свою очередь, содержат другие свойства. Например, свойствами-объектами являются свойства `Font` и `Constraints`. Значения таких свойств можно изменять через Инспектор объектов, кроме того, можно создать специальный редактор свойств, который будет открываться при нажатии в Инспекторе объектов кнопки с тремя точками.
- *Массивы* — содержат индексированную совокупность однотипных значений. Свойствами-массивами являются свойства `Lines` и `Items`. Значения этих свойств нельзя изменять через Инспектор объектов, для них нужно создать специальный редактор.

Для каждого свойства подключается соответствующий редактор, типы которых рассмотрены при изучении вопросов, связанных с использованием Инспектора объектов для построения интерфейсной части приложения.

Рассмотрим следующий пример компонента:

```
type  TBtnSound = class(TButton)
  private
     FCode: integer;
     FSign: char:
     FNote: string;
```

```
published
     property Code: integer read FCode write FCode;
     property Sign: char read FSign write FSign;
     property Note: string read FNote write FNote;
  end;
```

Здесь для каждого свойства описывается поле соответствующего типа. Все свойства доступны в Инспекторе объектов для чтения и для записи и непосредственно связаны со своими полями. У компонента есть свойства `Code` (целочисленное), `Sign` (символьное) и `Note` (строковое), доступные при работе с ним. По умолчанию свойства получают соответственно значения `0`, `#0` и `''` (пустая строка). Поскольку все свойства расположены в разделе опубликованных описаний, после установки компонента в Палитре компонентов и использовании в форме эти свойства становятся доступны разработчику через Инспектор объектов.

По своему функциональному назначению названные выше свойства похожи на свойство `Tag`, которое содержит значение, используемое и анализируемое самим программистом. В отличие, например, от свойства `Caption`, добавленные свойства (поля) внутри компонента его методами никак не используются, поэтому при изменении значений этих свойств изменений облика или поведения компонента не происходит.

Добавление перечислимого или множественного свойства не отличается от добавления простого свойства. При этом для описания можно использовать уже существующие или новые типы, которые следует предварительно определить.

Замечание

Логический тип, имеющий два возможных значения, относится к перечислимым типам.

Рассмотрим создание перечислимых и множественных свойств на примере. Пусть к компоненту добавляются перечислимые свойства `Sign`, `WeekDay` и `MyCursor`, а также множественные свойства `Frame` и `MyAnchors`. Для описания свойств и полей использованы существующие типы `boolean`, `TCursor` и `TAnchors` и новые типы `TWeekDay` и `TFrame`.

```
TWeekDay = (wdMonday, wdTuesday, wdWednesday, wdThursday,
            wdFriday, wdSaturday, wdSunday);
TFrame = set of (frTop, frBottom, frLeft, frRight);
TBtnSound = class(TButton)
  private
         FSign:   boolean;
      FWeekDay:   TWeekDay;
     FMyCursor:   TCursor;
        FFrame:   TFrame;
    FMyAnchors:   TAnchors;
public
     property Sign: boolean read FSign write FSign;
     property MyAnchors: TAnchors read FMyAnchors write FMyAnchors;
published
     property WeekDay: TWeekDay read FWeekDay write FWeekDay;
     property MyCursor: TCursor read FMyCursor write FMyCursor;
     property Frame: TFrame read FFrame write FFrame;
end;
```

Все свойства позволяют читать и записывать свои значения. Однако программно доступны только свойства `WeekDay`, `MyCursor` и `Frame`, описанные в разделе общедоступных описаний.

Для описания свойства-объекта можно использовать уже существующий тип какого-либо объекта, например, `TFont` или новый тип. Описание типа объекта представляет собой описание класса, предком которого являются классы `TPersistent` и его потомки.

Перед использованием нового объекта он должен быть предварительно создан, а после применения удален. Ответственность за эти операции несет класс, использующий этот объект, поэтому в данном классе необходимо переопределить методы создания и удаления.

Рассмотрим пример создания свойства-объекта (листинг 39.6). Пусть к классу `TBtnSound` добавляется свойство-объект `MyObject` типа `TMyObject`. Класс `TMyObject` описывается перед классом `TBtnSound` и, в свою очередь, имеет три простых свойства: `ObjectNumber`, `ObjectAttrib` и `ObjectName`. Из них свойство `ObjectNumber` доступно для чтения, а свойство `ObjectName` — только для записи.

Листинг 39.6. Пример создания свойства-объекта

```
type
TMyObject = class(TPersistent)
  private
    FObjectNumber: integer;
    FObjectAttrib: boolean;
      FObjectName: string;
  published
    property ObjectNumber: integer read FObjectNumber;
    property ObjectAttrib: boolean read FObjectAttrib write FObjectAttrib;
    property ObjectName: string write FObjectName;
    end;
TBtnSound = class(TButton)
  private
     FMyObject: TMyObject;
  public
     constructor Create(AOwner: TComponent); override;
     destructor Destroy; override;
  published
     property MyObject: TMyObject read FMyObject write FMyObject;
  end;
...
constructor TBtnSound.Create(AOwner: TComponent);
begin
  inherited;
  FMyObject := TMyObject.Create;
end;

destructor TBtnSound.Destroy;
begin
  FMyObject.Free;
  inherited;
end;
```

В конструкторе класса TBtnSound после создания объекта этого класса создается и объект TMyObject, а в деструкторе класса TBtnSound перед удалением объекта этого класса происходит удаление объекта TMyObject. В обоих методах с помощью инструкции inherited; вызывается унаследованный метод предка. При этом важна последовательность вызова методов: в конструкторе унаследованный метод вызывается до создания объекта TMyObject, а в деструкторе — уже после удаления этого объекта.

В случае, когда для чтения и записи значений свойств используются методы, программист должен эти методы создать, например, как показано в листинге 39.7.

Листинг 39.7. Пример методов для чтения и записи свойств

```
TBtnSound = class(TButton)
  private
                FCode2: integer;
     function GetCode2: integer;
     procedure SetCode2(var pCode2 :integer);
 published
     property Code2: integer read GetCode2 write SetCode2;
 end;
...
function TBtnSound.GetCode2: integer;
begin
  if FCode2 > 0 then Result := FCode2 else Result := 0;
end;

procedure TBtnSound.SetCode2(var pCode2: integer);
begin
  if FCode2 <> pCode2 then FCode2 := pCode2;
end;
```

К классу TBtnSound добавляется простое свойство Code2 целочисленного типа. Для доступа к его значению используются методы GetCode2 и SetCode2.

Создание и переопределение методов

Новые методы *создаются* с учетом назначения компонента и придают компоненту новую функциональность. Методы, добавляемые к классу нового компонента, не отличаются от обычных методов класса, которые были рассмотрены в *главе 2*, посвященной языку Delphi. Отметим, что эти методы имеют доступ к полям класса и могут выполнять операции, связанные с чтением и записью значений полей.

Переопределение метода нового компонента выполняется в случае, когда этот метод имеет то же имя, что и метод предка, но выполняет отличающиеся действия. Для этого в разделе защищенных описаний класса указывается заголовок этого метода, а после него ставится модификатор override, означающий, что метод предка в данном классе переопределяется. В разделе реализации модуля приводится новый код метода.

В ряде случаев нет необходимости переопределять весь код метода предка, а достаточно только изменить некоторые действия или дополнить их. Тогда в теле метода снача-

ла вызывается метод предка, после чего кодируются нужные операции, выполняющие дополнительные действия.

Пусть, например, для кнопки `BtnSound` нужно переопределить метод `Click`. В результате при нажатии кнопки, кроме привычных действий, будет издаваться системный звук.

```
TBtnSound = class(TButton)
    protected
    procedure Click; override;
end;
...
procedure TBtnSound.Click;
begin
  Beep;
  inherited;
end;
```

Переопределение методов, связанных с обработкой событий, также называют *переопределением событий*.

Создание значка компонента

По умолчанию компонент получает от своего предка значок, который отображается в Палитре компонентов. Для создания значка нового компонента вызывается редактор ресурсов Image Editor и создается новый файл ресурса Component Resource File с расширением dcr. Формат dcr-файла ресурсов совпадает с форматом res-файла ресурсов.

В dcr-файл ресурсов добавляется растровый рисунок, который первоначально имеет имя `Bitmap1`; ему необходимо присвоить имя, совпадающее с именем класса нового компонента и написанное заглавными буквами, например, `TBTNSOUND`. Рисунок должен иметь размеры 24×24 пиксела и 16 или 256 цветов (обычно выбирается вариант 16 цветов). При сохранении файлу ресурса нужно присвоить имя, совпадающее с именем модуля компонента, например Btnsound.dcr, и поместить в тот же каталог.

Инсталляция компонента

После подготовки модуля компонента и сохранения его на диске компонент можно инсталлировать — поместить в Палитру компонентов. Инсталляцию компонента можно выполнить несколькими способами, например, с помощью команды **Component | Install Component**, которая открывает диалоговое окно **Install Component** (рис. 39.5).

В поле **Unit file name** указывается имя файла модуля, в котором находится инсталлируемый компонент. По умолчанию предлагается модуль, который отображается в редакторе кода. Поле **Search path** содержит список каталогов, в которых выполняется поиск файлов, необходимых для инсталляции компонента.

Каждый компонент помещается в пакет. Первоначально предполагается использование существующего пакета, заданного в поле описания пакета **Package description**. Если для компонента нужно создать новый пакет, то его название и описание следует ука-

зать на вкладке **Into new package**. По умолчанию новые компоненты помещаются в пользовательский пакет dclusr.dpk.

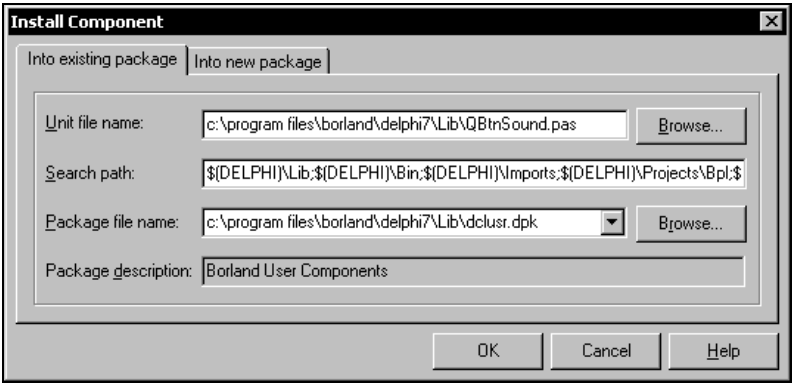


Рис. 39.5. Диалоговое окно инсталляции компонента

После нажатия кнопки **OK** выполняется инсталляция компонента, в процессе которой запрашивается подтверждение на внесение изменений в указанный пакет. Если инсталляция прошла без ошибок, то выдается соответствующее сообщение, а новый компонент появляется на указанной странице Палитры компонентов, и его можно использовать при разработке приложений.

При инсталляции на экране появляется окно редактора пакетов **Package** (рис. 39.6), в котором отображаются название и содержимое изменяемого пакета, в данном случае dclusr.dpk. С помощью этого редактора можно добавить или удалить отдельный элемент пакета, а также выполнить компиляцию модуля компонента и инсталляцию компонента.

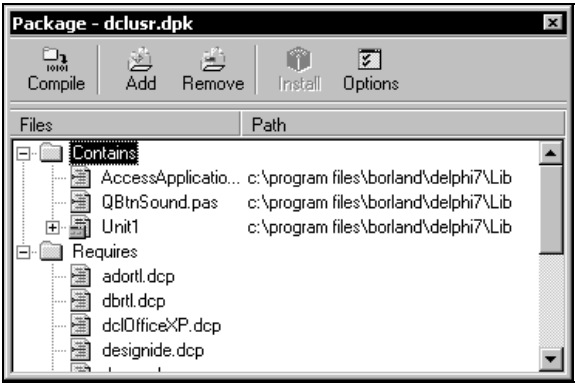


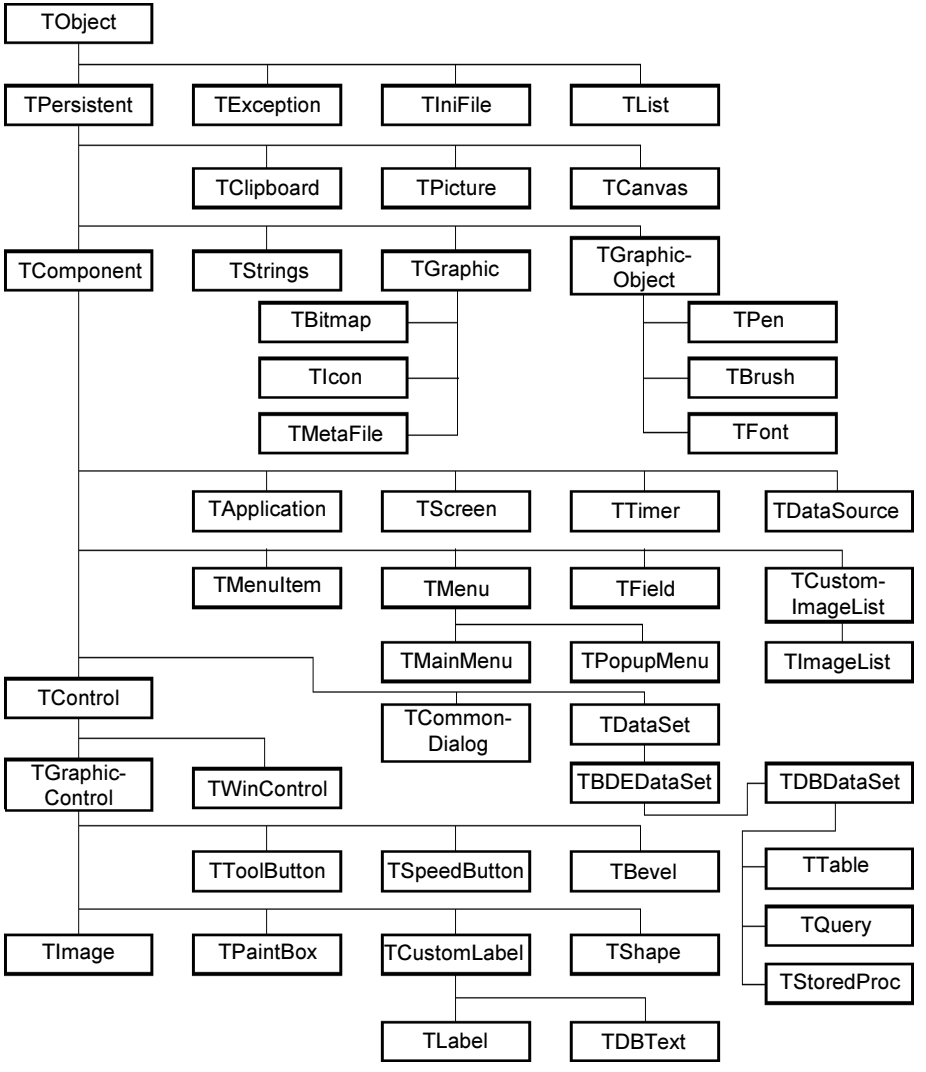
Рис. 39.6. Окно редактора пакетов

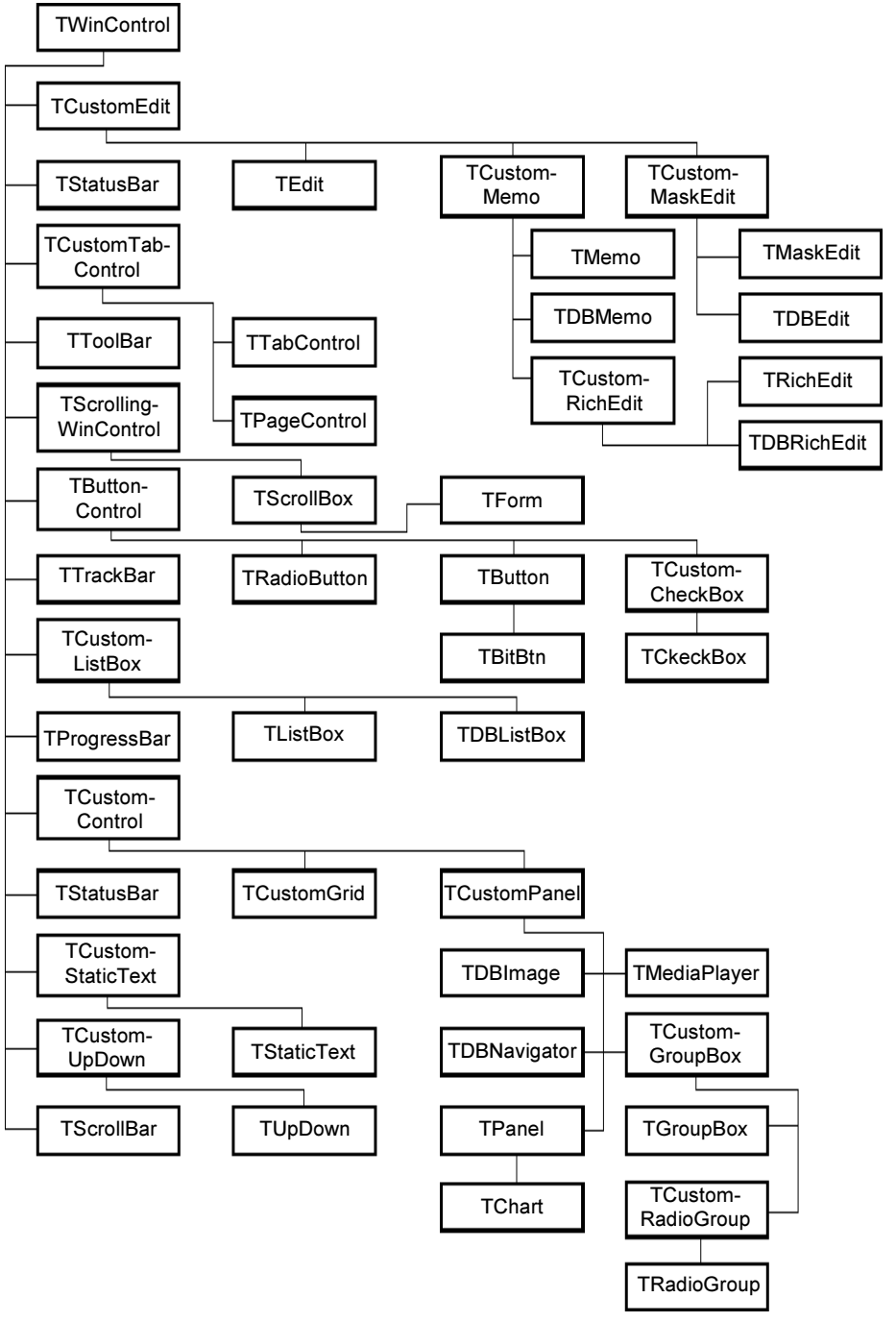
Замечание

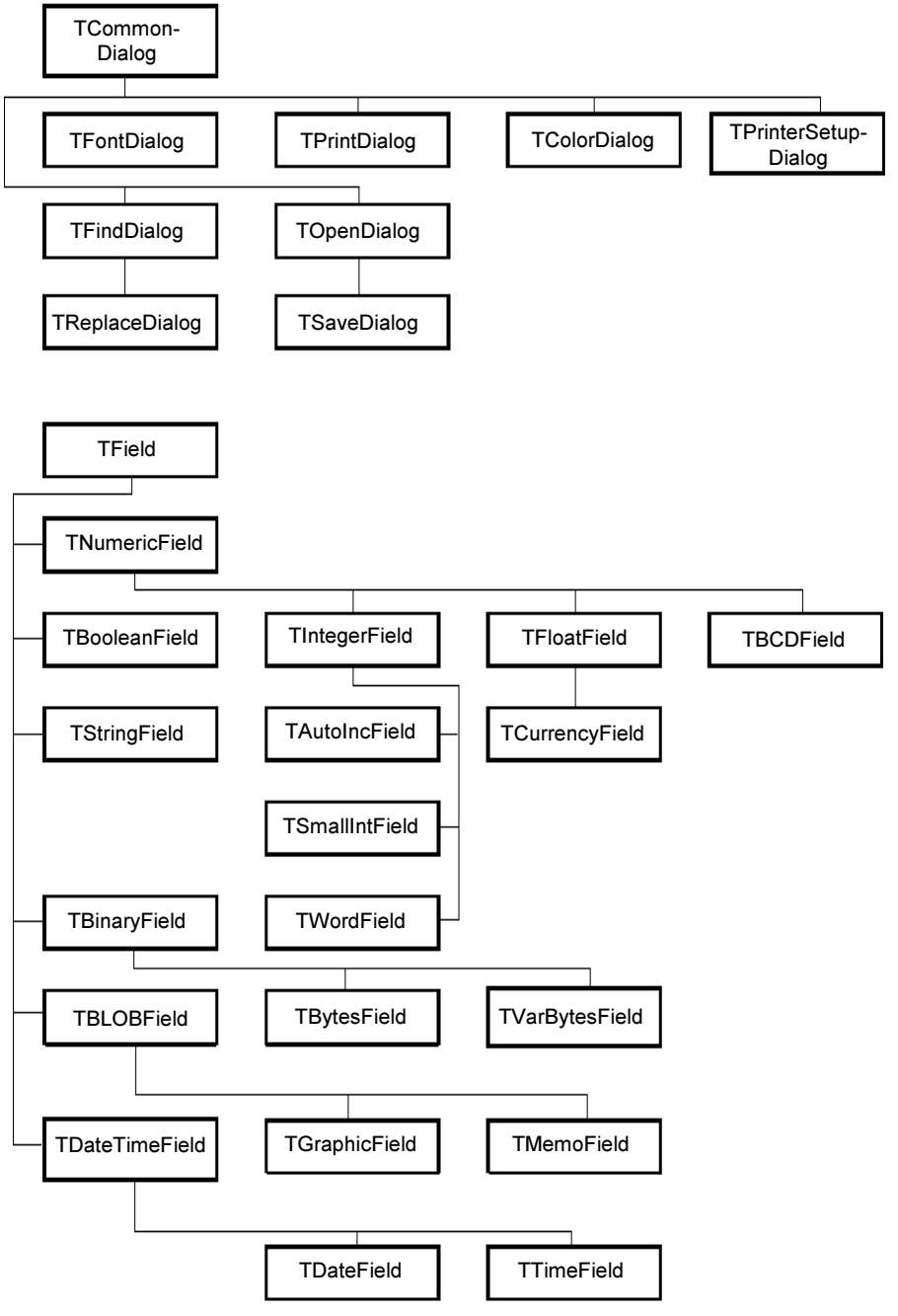
Инсталляцию компонента можно выполнять только после сохранения файла модуля на диске.

ПРИЛОЖЕНИЕ 1

Фрагменты иерархии классов VCL




TWinControl
TCustomEdit
TStatusBar
TEdit
TCustom-Memo
TCustom-MaskEdit
TCustomTab-Control
TMemo
TMaskEdit
TDBMemo
TDBEdit
TToolBar
TTabControl
TCustom-RichEdit
TRichEdit
TScrolling-WinControl
TPageControl
TDBRichEdit
TButton-Control
TScrollBox
TForm
TTrackBar
TRadioButton
TButton
TCustom-CheckBox
TCustom-ListBox
TBitBtn
TCkeckBox
TProgressBar
TListBox
TDBListBox
TCustom-Control
TStatusBar
TCustomGrid
TCustomPanel
TCustom-StaticText
TDBImage
TMediaPlayer
TCustom-UpDown
TStaticText
TDBNavigator
TCustom-GroupBox
TScrollBar
TUpDown
TPanel
TGroupBox
TChart
TCustom-RadioGroup
TRadioGroup


TCommon-Dialog
TFontDialog
TPrintDialog
TColorDialog
TPrinterSetup-Dialog
TFindDialog
TOpenDialog
TReplaceDialog
TSaveDialog
TField
TNumericField
TBooleanField
TIntegerField
TFloatField
TBCDField
TStringField
TAutoIncField
TCurrencyField
TSmallIntField
TBinaryField
TWordField
TBLOBField
TBytesField
TVarBytesField
TDateTimeField
TGraphicField
TMemoField
TDateField
TTimeField

ПРИЛОЖЕНИЕ 2

Описание компакт-диска

На компакт-диске в папках с именами соответствующих глав содержатся листинги примеров из книги. Все листинги представляют либо фрагменты исходных кодов, либо полные исходные коды приложений, которые отлаживались в среде Delphi 7. На их основе нужно создать приложение Delphi, поместить на форму приложения компоненты, описания которых приведены в примере, настроить свойства компонентов с помощью Инспектора объектов. Чтобы определить функциональность приложения, достаточно скопировать с компакт-диска все имеющиеся процедуры — обработчики событий согласно рекомендациям из раздела *главы 1* о разработке приложения.

Предметный указатель

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

А

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Х

Ш

Э

Я