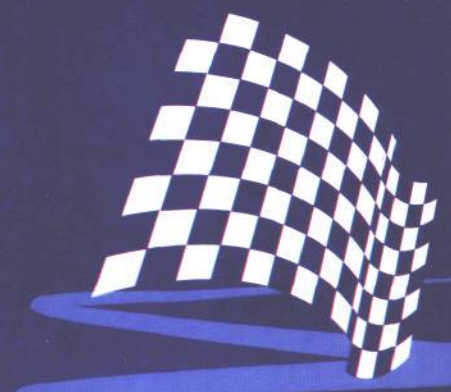
ВЛАДИМИР ГОФМАН
АНАТОЛИЙ ХОМОНЕНКО

www.bhv.ru
www.bhv.kiev.ua

Delphi

Быстрый старт

- Основы работы в среде Delphi
- Приемы создания приложений
- Разработка и использование баз данных
- Примеры программ



Содержание

Владимир Гофман

Анатолий Хомоненко

Delphi

Быстрый старт

Санкт-Петербург
«БХВ-Петербург»
2003

УДК 681.3.06
ББК 32.973.26-018
Г74

Гофман В. Э., Хомоненко А. Д.

Г74 Delphi. Быстрый старт. — СПб.: БХВ-Петербург, 2003. — 288 с: ил.

ISBN 5-94157-165-8

В книге описываются интерфейс системы визуального программирования Delphi на основе 6-й версии, состав и характеристика элементов проекта приложения, приемы программирования на языке Object Pascal. Рассматриваются визуальные компоненты, используемые для создания интерфейса приложений; техника работы с текстовой информацией, кнопками и переключателями, а также формами, являющимися центральной частью любого приложения; создание меню. Даются понятия, используемые в теории баз данных; обсуждаются элементы реляционных баз данных и особенности их использования; описываются создание таблиц и приложения баз данных, приемы работы с данными, подготовка отчетов.

Для начинающих программистов

УДК 681.3.06
ББК 32.973.26-018

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Анна Кузьмина*
Компьютерная верстка	*Татьяны Соколовой*
Корректор	*Татьяна Звертановская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.12.02.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 23,22.
Доп. тираж 5000 экз. Заказ № 641
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г, выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-165-8

Предисловие

В настоящее время среди широкого круга пользователей популярна система объектно-ориентированного программирования Delphi, основу которой составляет язык Object Pascal. Delphi позволяет быстро создавать приложения различной степени сложности на основе применения технологии визуального программирования.

Книга посвящена основам работы с Delphi и освоению приемов программирования с использованием визуальных средств. В ней рассматриваются важнейшие средства Delphi, технология создания приложений для решения общих задач (от простейших программ до приложений среднего уровня сложности, предназначенных для работы с базами данных), с которыми приходится сталкиваться на начальном этапе освоения системы программирования.

В книге рассматривается большое число примеров, демонстрирующих основные возможности Delphi. Примеры взяты из работающих программ, которые читатель может использовать в своих проектах. Нами рассматривается использование наиболее общих языковых средств, присутствующих в последних версиях системы. Как следствие, книга не привязана к конкретной версии и приведенные примеры работоспособны для 4-й–6-й версий Delphi. При описании интерфейса использовалась последняя на текущее время 6-я версия системы.

Книга включает две части.

Часть I. Основные средства Delphi. Содержит описание интерфейса системы программирования, состав и характеристику элементов проекта приложения. Описывается язык программирования Object Pascal: типы данных, основные конструкции языка, важнейшие приемы программирования, понятия объектно-ориентированного программирования.

Рассматриваются важнейшие визуальные компоненты, используемые для создания интерфейса приложений. При этом дается состав страниц Палитры компонентов, содержащих визуальные компоненты; описывается класс `TControl`, который является базовым для большинства визуальных компонентов и включает в себя общие для визуальных компонентов свойства, события и методы.

Рассматриваются компоненты и техника работы с информацией (текстом) по ее отображению, вводу и редактированию. С этой целью описывается класс Tstrings, являющийся базовым классом для операций со строковыми данными; компонент Label, служащий для отображения надписей (текста, используемого в качестве заголовков для некоторых управляющих элементов); компоненты Edit, `MaskEdit`, Memo и RichEdit, обеспечивающие возможности редактирования информации; средства и техника работы со списками.

Обсуждаются компоненты и техника работы с кнопками и переключателями. Освещается техника объединения, или группирования различных элементов управления, которая может понадобиться, например, при работе с переключателями на форме или при создании панели инструментов.

Рассматриваются компоненты и техника создания форм, являющихся важнейшим визуальным компонентом, центральной частью практически любого приложения и представляющих собой видимые окна Windows. Приводятся характеристики формы, приемы организации взаимодействия форм.

Описываются компоненты и техника работы с меню (главным и контекстным), которое является распространенным элементом пользовательского интерфейса приложения и служит для управления его работой.

Часть II. Работа с базами данных. Рассматриваются основные понятия баз данных; характеризуются элементы реляционных баз данных и техника их использования (таблицы, ключи и индексы, способы доступа к данным, связь между таблицами, механизм транзакций и др.); дается обзор средств и компонентов для работы с базами данных; характеризуется технология создания информационной системы; обсуждается создание таблиц и приложения баз данных.

Описываются основные компоненты для работы с данными: доступа к данным и визуальные компоненты. При этом рассматриваются общие свойства наборов данных, используемых для выполнения операций над данными таблиц. Освещаются состояния и режимы наборов данных, доступ к полям. Отмечаются особенности важнейших наборов данных Table и Query. Описываются невизуальные объекты типа `TField`, служащие для доступа к данным полей записей набора данных, работа с Редактором полей и доступ к значениям полей. Освещается использование источника данных, служащего промежуточным звеном между набором данных и визуальными компонентами. Рассматриваются сами визуальные компоненты, с помощью которых пользователь управляет набором данных (используются для навигации по набору данных, а также для отображения и редактирования записей).

Работа с данными рассматривается на примере использования навигационного способа доступа к локальным БД, допускающего использование наборов данных Table или Query. При этом освещаются важнейшие операции, реализуемые при навигационном способе доступа: сортировка записей в наборе данных, навигация по набору данных, редактирование записей, вставка и удаление записей, фильтрация записей. Здесь же описывается работа со связанными таблицами.

Рассматривается подготовка отчетов — печатных документов, содержащих данные, аналогичные получаемым в результате выполнения запроса к базе данных. При этом описываются компоненты, предназначенные для создания отчетов; процедура печати отчета; технология подготовки простого отчета.

Книга ориентирована на начинающих пользователей.

Часть I

Основные средства Delphi

Глава 1

Среда Delphi

Создание прикладных программ, или приложений, Delphi выполняется в интегрированной среде разработки IDE (Integrated Development Environment). IDE служит для организации взаимодействия с программистом и включает в себя ряд окон, содержащих различные управляющие элементы. С помощью средств интегрированной среды разработчик может удобно проектировать интерфейсную часть приложения, а также писать программный код и связывать его с управляющими элементами. При этом вся работа по созданию приложения, включая отладку, происходит в интегрированной среде разработки.

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

- главное окно **(Delphi 6 — Projectl);**
- окно Обозревателя дерева объектов **(Object TreeView)**;
- окно Инспектора объектов **(Object Inspector);**
- окно Конструктора формы (**Form1**);
- окно Редактора кода **(Unitl.pas);**
- окно Проводника кода **(Exploring Unitl.pas).**

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

Несмотря на наличие многих окон, Delphi является однодокументной средой и позволяет одновременно работать с одним приложением (проектом

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

Замечание

Приведенный вид интегрированной среды соответствует 6-й версии Delphi. Интегрированная среда других версий имеет небольшие отличия. В частности, в 5-й версии отсутствует окно Обозревателя дерева объектов.

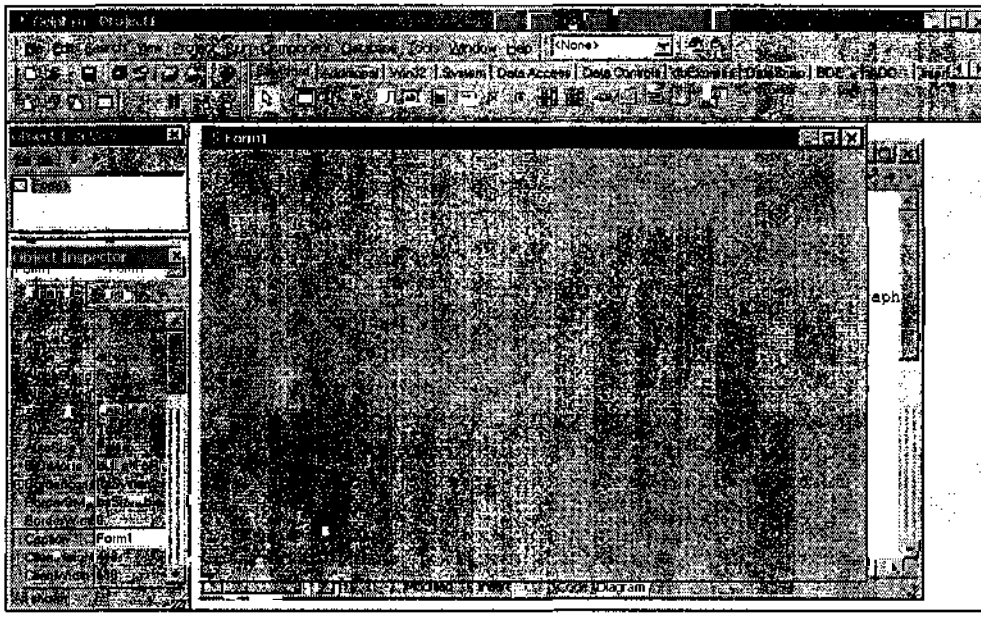


Рис. 1.1. Вид интегрированной среды разработки

При минимизации главного окна происходит минимизация всего интерфейса Delphi и соответственно всех открытых окон, 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 (Выполнение | Выполнить) можно вызвать с помощью клавиши <F9>, а команду View | Units (Просмотр | Модуль) — с помощью комбинации клавиш <Ctrl>+<F12>.

Всего имеется 6 панелей инструментов:

- Standard (Стандартная);
- View (Просмотра);
- Debug (Отладки);
- Custom (Пользователя);
- Desktop (Рабочий стол);
- Internet (Интернет).

Можно управлять отображением панелей инструментов и настраивать состав кнопок на них. 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 Component Palette (Палитры компонентов).

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 иконками. Нужная страница Палитры компонентов выбирается щелчком мышью на ее ярлычке. К числу основных страниц Палитры компонентов можно отнести следующие:

- Standard (Стандартная);
- Additional (Дополнительная);
- Win32 (32-разрядный интерфейс Windows);
- System (Доступ к системным функциям);
- Data Access (Работа с информацией из баз данных);
- **Data Controls** (Создание элементов управления данными);
- BDE (Доступ к данным с помощью BDE (в среде 6-й версии));
- **QReport** (Составление отчетов);
- Dialogs (Создание стандартных диалоговых окон).

Окно Конструктора формы первоначально находится в центре экрана и имеет заголовок **Form1**. В нем выполняется проектирование формы, для

чего на форму из Палитры компонентов помещаются необходимые компоненты. При этом проектирование заключается в визуальном конструировании формы, а работа разработчика похожа на работу в среде простого графического редактора. Сам Конструктор формы во время ее проектирования остается "за кадром", и разработчик имеет дело с самой формой, поэтому часто окно Конструктора также называют окном формы или просто формой.

Окно Редактора кода (заголовок **Unit1.pas**) после запуска системы программирования находится под окном Конструктора формы и почти полностью перекрывается им. Редактор кода (Редактор) 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 на отдельной странице, доступ к которой осуществляется щелчком на соответствующем ярлычке. Первоначально в окне Редактора кода на странице **Code** содержится одна закладка **Unit1** исходного кода модуля формы `Form1` разрабатываемого приложения.

Переключение между окнами Конструктора формы и Редактора кода удобно выполнять с помощью клавиши <F12>.

Окно Проводника кода **(Exploring Unitl.pas)** пристыковано слева от окна Редактора кода. В нем в виде дерева отображаются все объекты модуля формы, например, переменные и процедуры (рис. 1.2). В окне Проводника кода можно удобно просматривать объекты приложения и быстро переходить к нужным объектам, что особенно важно для больших модулей. Вызов окна Проводника кода выполняется по команде **Code Explorer** (Проводник кода) меню **View** (Вид).

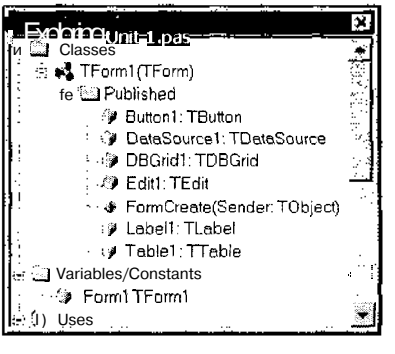


Рис. 1.2. Окно Проводника кода

Окно Инспектора объектов находится в левой части экрана и отображает свойства и события объектов для текущей формы Formi. Его можно вызвать

на экран 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 Конструктора 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 существует 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* или *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 Инспектор объектов. Компонент, расположенный на форме, можно выбрать щелчком мыши на нем или выбором в списке Инспектора объектов. Каждый компонент имеет свой набор свойств и событий, определяющих его особенности.

Начиная с 4-й версии, Delphi поддерживает технологию *Dock-окон,* которые могут стыковаться (соединяться) друг с другом с помощью мыши. *Стыкующимися* окнами являются инструментальные (не диалоговые) окна интегрированной среды разработки, в том числе окна Инспектора объектов и Проводника кода. Со стыкованными окнами удобнее выполнять такие операции, как перемещение по экрану или изменение размеров.

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 от второго окна. 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/расстыковке окно изменяет свое название. Так, окно Проводника кода, состыкованное с окном Редактором кода, имеет общее с ним название, например, **Unitl.pas,** в то время как при отстыковке название изменяется на **Exploring Unitl.pas.** Окна Инспектора объектов и Обозревателя дерева объектов при стыковке объединяют свои названия (через запятую указываются названия каждого из окон).

Скрытое окно вызывается на экран командой пункта **View** (Просмотр) главного меню. Например, окно Проводника кода выводится на экран командой **View | Code Explorer** (Просмотр | Проводник кода).

1.1. Характеристика проекта

1.1.1. 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);
- модули форм (PAS);
- модули (PAS);
- параметры проекта (DOF);
- описание ресурсов (RES).

Взаимосвязи между отдельными частями (файлами) проекта показаны на рис. 1.3.

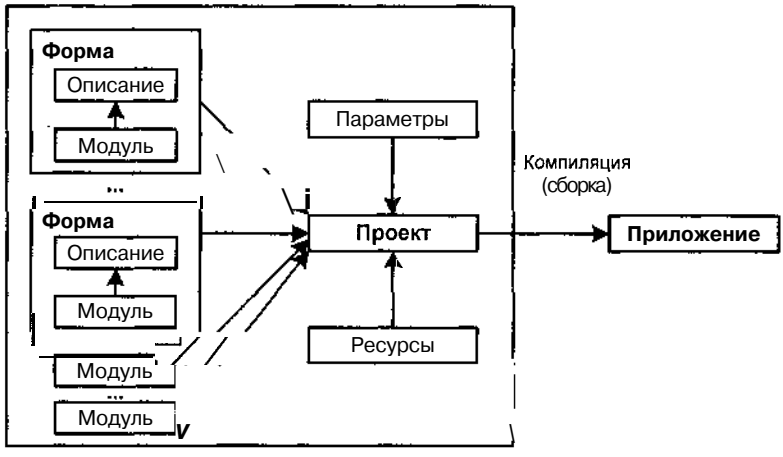


Рис. 1.3. Связь между файлами проекта

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 Project1, имя которого отображается в заголовке главного окна Delphi. Этот проект имеет в своем составе одну форму Form1, название которой видно в окне Конструктора формы. Разработчик может изменить проект, предлагаемый по умолчанию, а также установить параметры среды, при которых после загрузки Delphi будет загружаться приложение, разработка которого выполнялась в последний раз.

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

1.1.2. Файл проекта

Файл проекта является центральным файлом проекта и представляет собой собственно программу. Для приложения, включающего в свой состав 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 автоматически переименовываются.

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 будем предполагать, 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, который по умолчанию имеет имя, совпадающее 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.

Программа проекта содержит всего три оператора, выполняющих инициализацию приложения, создание формы Formi и запуск приложения. Эти операторы рассмотрены в последующих главах.

При выполнении разработчиком каких-либо операций с проектом код файла проекта формируется Delphi 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 также автоматически исключаются. При необходимости программист может вносить изменения в файл проекта самостоятельно, однако подобные действия могут разрушить целостность проекта и поэтому обычно выполняются опытными программистами. Отметим, что ряд операций, например, создание обработчика события для объекта Application не может быть выполнен Delphi автоматически и требует самостоятельного кодирования в файле проекта.

Для отображения кода файла проекта в окне Редактора кода с целью просмотра и редактирования достаточно выполнить команду **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.

1.1.3. Файлы формы

Для каждой формы в составе проекта автоматически создаются файл описания (DFM) и файл модуля (PAS).

Файл описания формы является ресурсом Delphi и содержит характеристики формы и ее компонентов. Разработчик обычно управляет этим файлом через окно Конструктора формы и Инспектор объектов. При конструировании формы в файл описания автоматически вносятся соответствующие изменения. 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 форма пропадает с экрана, а содер-

жимое файла ее описания открывается в окне Редактора кода и доступно для просмотра и редактирования.

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. Повторное открытие окна Конструктора формы выполняется командой **View | Forms** (Просмотр | Формы) или комбинацией клавиш <Shift>+<F12>, в результате чего открывается диалоговое окно **View Form** (Просмотр форм) (рис. 1.4), в котором выбирается нужная форма.

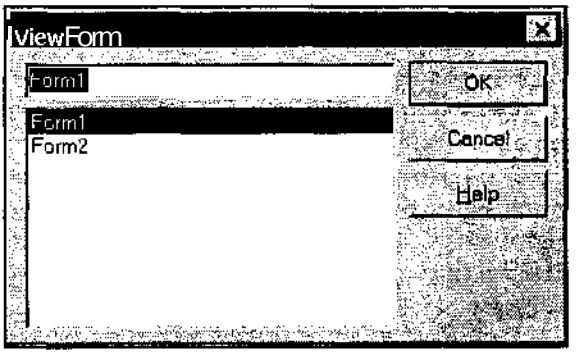


Рис. 1.4. Выбор формы для открытия

Одновременно можно отобразить на экране несколько форм. Для закрытия окна Конструктора той или иной формы достаточно 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 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.
```

Файл модуля формы создается Delphi автоматически 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 на форме компонентов, а также при создании обработчиков событий в модуль формы вносятся соответствующие изменения. При этом часть этих изменений вносится Delphi автоматически, а другую часть вносит разработчик. Обычно все действия разработчика, связанные с программированием, выполняются именно в модулях форм.

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

Отметим, что оба файла каждой формы (описания и модуля) имеют одинаковое имя, которое отличается от имени файла проекта, несмотря на то, что файл проекта имеет другое расширение.

При компиляции модуля автоматически создается файл с расширением DCU, который содержит откомпилированный код модуля. Этот файл можно удалять из каталога, в котором находятся все файлы проекта, но Delphi снова создает этот файл при следующей компиляции модуля или проекта.

1.1.4. Файлы модулей

При программировании, кроме модулей в составе форм, можно использовать *отдельные модули,* не связанные с какой-либо формой. Они оформля-

ются 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 целесообразно размещать процедуры, функции, константы и переменные, общие для нескольких модулей проекта.

1.1.5. Файл ресурсов

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

- пиктограммы;
- растровые изображения;
- курсоры.

Перечисленные компоненты являются ресурсами Windows. Первоначально файл ресурса содержит пиктограмму проекта, которой по умолчанию является изображение факела.

Для работы с файлами ресурсов в состав Delphi включен графический редактор Image Editor версии 3.0, вызываемый командой **Tools | Image Editor** (Средства | Редактор изображений).

1.1.6. Параметры проекта

Для установки параметров проекта используется окно параметров проекта **(Project Options),** вызываемое командой **Project | Options** (Проект | Параметры) или нажатием комбинации клавиш <Ctrl>+<Shift>+<FH>. В частности, в этом окне можно задать главную форму приложения, задать справочный файл или сменить пиктограмму приложения.

1.2. Компиляция и выполнение проекта

В процессе компиляции проекта создается готовый к использованию файл, которым может быть приложение (EXE) или динамически загружаемая библиотека (DLL). Далее будем рассматривать только файл-приложение. 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.

Запуск процесса компиляции выполняется по команде **Project | Compile <Projectl>** (Проект | Компилировать <проект>) или нажатием комбинации

клавиш <Ctrl>+<F9>. В этой команде содержится имя проекта, разработка которого выполняется в настоящий момент, первоначально это Project1. При сохранении проекта под другим именем соответственно должно измениться имя проекта в команде меню.

Компиляция проекта для получения приложения может быть выполнена на любой стадии разработки проекта. Это удобно для проверки вида и правильности функционирования отдельных компонентов формы, а также для проверки отдельных фрагментов создаваемого кода. При компиляции проекта выполняются следующие действия:

- 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 использующие его с помощью директивы uses МОДУЛИ;
- перекомпиляция модуля происходит также при изменениях объектного файла (OBJ) или подключаемого файла (INC), используемых данным модулем;
- после компиляции всех модулей проекта компилируется файл проекта и создается исполняемый файл приложения с именем файла проекта.

Кроме компиляции может быть выполнена *сборка* проекта. При сборке компилируются все файлы, входящие в проект, независимо от того, были в них внесены изменения или нет. Для сборки проекта используется команда меню **Project | Build <Project1>** (Проект | Собрать <проект1>).

Запустить проект на выполнение можно из среды Delphi и из среды Windows.

Выполнение проекта *из среды Delphi* осуществляется командой **Run | Run** (Выполнение | Выполнить) 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 | Остановить программу) или комбинации клавиш <Ctrl>+<F2>.

Для отладки приложений в среде Delphi можно использовать средства отладчика.

Из среды Windows созданное приложение можно запустить как и любое другое приложение, например, с помощью Проводника.

1.3. Разработка приложения

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

- создание интерфейса приложения;
- определение функциональности приложения.

Интерфейс приложения определяет способ взаимодействия пользователя и приложения, то есть внешний вид формы (форм) при выполнении приложения, и то, каким образом пользователь управляет приложением. Интерфейс создается путем размещения в форме компонентов, которые называются *интерфейсными* или *управляющими* компонентами (элементами). Создание интерфейса приложения выполняется с помощью Конструктора формы.

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

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

1.3.1. Простейшее приложение

Для примера создадим простейшее приложение. Слово "создадим" в этом случае является несколько громким, так как создавать и тем более программировать не придется вообще ничего. Delphi уже изначально представляет готовое приложение, состоящее из одной формы.

Непосредственно после начала 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 минимизации, максимизации и закрытия окна, изменения размеров окна и кнопку вызова системного меню окна.

Именно эта форма отображается при первом запуске Delphi в окне Конструктора формы.

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

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

Простейшее приложение представляет из себя заготовку или каркас, обеспечивающий разработчика всем необходимым каждому приложению вообще. Так, не нужно писать свой обработчик 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 полностью сделано создателями Delphi, и каркас приложения представляет собой полностью завершенное и функционирующее приложение, которое просто "ничего не делает".

Фразу "ничего не делает" можно понимать двояко. Окно, а вместе с ним и приложение, действительно ничего не делает в том смысле, что не имеет функциональности, специфичной для каждого приложения. Вместе с тем, это пустое окно выполняет достаточно большую работу. 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 старым традиционным способом. Изнутри Windows представляет систему с индексами, контекстами, обратными вызовами и множеством других сложнейших элементов, которые нужно знать, которыми нужно управлять и в которых можно легко запутаться. Поскольку эти элементы имеются в каждом функционирующем приложении Windows, система Delphi скрывает эти сложности от программиста. О той работе, которую проделывает Delphi за программиста, можно судить также по размеру полученного выполнимого файла простейшего приложения (для 6-й версии примерно 355 Кбайт).

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

1.3.2. Создание интерфейса приложения

Интерфейс приложения составляют компоненты, которые разработчик выбирает из Палитры компонентов и размещает на форме, сами компоненты являются своего рода строительными блоками. При конструировании интерфейса приложения действует принцип WYSIWYG (What You See Is What You Get, что видите, то и получите), 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). Понятие 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 форма. Так как визуальные компоненты используются пользователем для управления приложением, то эти компоненты также называют *управляющими компонентами* или элементами управления. Именно визуальные компоненты образуют 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 записи таблицы базы данных.

При создании интерфейса приложения для каждого компонента выполняются следующие операции:

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

Разработчик выполняет эти операции в окне Конструктора 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 весь процесс создания приложения называют не программированием, а конструированием.

Выбор компонента в палитре выполняется щелчком мыши на нужном компоненте, например, кнопке Button, в результате чего его пиктограмма принимает утопленный вид. Если после этого щелкнуть на свободном месте формы, то на ней появляется выбранный компонент, а его пиктограмма в палитре принимает обычный вид. Пиктограммы 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 подсказка о его назначении.

В обозначении типа объектов Delphi, в том числе и компонентов, указывается буква т. Часто для обозначения компонентов используются не их названия, а типы. Для обозначения компонентов будем использовать именно названия, а не типы компонентов, то есть Button, а не TButton, Label, а не TLabel.

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

```
<Название компонента>: <Тип компонента>;
```

Название компонента является значением его свойства Name, а тип определяется выбранным компонентом. Например, для кнопки Button эта строчка первоначально будет иметь вид:

```
Button1: TButton;
```

Для размещения на 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 пиктограмма в палитре остается утопленной, и каждый последующий щелчок на форме приводит к появлению на ней еще одного такого же компонента. Для отмены выбора данного компонента достаточно выбрать другой компонент или щелкнуть мышью на изображении стрелки в левом краю Палитры компонентов.

После размещения компонента в форму с помощью мыши можно изменять его положение и размеры. Кроме того, для 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. Одновременное выделение на форме нескольких компонентов можно выполнить щелчком мыши при нажатой клавише <Shift>.

Внешний вид компонента определяют его свойства, которые доступны в окне Инспектора объектов, когда компонент выделен на форме и вокруг него отображаются маркеры выделения (рис. 1.5). 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.

В ниспадающем списке, расположенном в верхней части окна Инспектора объектов, отображается название компонента и его тип. Выбрать тот или иной компонент и соответственно получить доступ к его свойствам также можно, выбрав этот компонент в списке Инспектора объектов. Такой способ выбора удобен в случаях, когда компонент полностью закрыт другими объектами.

В левой части окна Инспектора объектов приводятся названия всех свойств компонента, которые доступны на этапе разработки приложения. Для каж-

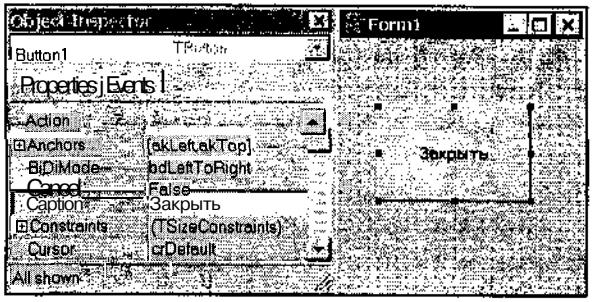


Рис. 1.5. Доступ к свойствам компонента

дого свойства справа содержится значение этого свойства. Отметим, что, кроме этих свойств, компонент может иметь свойства, которые доступны только *во время выполнения* приложения.

Свойства представляют собой атрибуты, определяющие способ отображения и функционирования компонентов при выполнении приложения. Каждый компонент имеет значения свойств по умолчанию. После помещения в форму компонента 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 Конструктора формы ("рисование") или с помощью Инспектора объектов.

Разработчик может изменить значение свойства компонента, введя или выбрав нужное значение. При этом одновременно изменяется соответствующий компонент, таким образом уже при проектировании видны результаты сделанных изменений. Например, при изменении значения свойства `Caption` (название) кнопки в процессе редактирования нового названия на поверхности кнопки отображается редактируемое название.

Для утверждения нового значения свойства достаточно нажать клавишу <Enter> или перейти к другому свойству или компоненту. Для отмены изменений необходимо нажать клавишу <Esc>. Если свойству введено неправильное значение, то выдается предупреждающее сообщение, а изменения значения отвергаются. Изменения свойств автоматически учитываются в файле описания формы, используемом компилятором при создании формы, а при изменении свойства Name вносятся изменения и в описание класса формы.

Каждый компонент для большинства своих свойств, например, `Color` (Цвет), `Caption` (Заголовок) и `visible` (Видимость), имеет значения по умолчанию.

Для обращения к компоненту в приложении предназначено свойство Name типа TComponentName, которое указывает имя компонента. Отметим, что тип

TComponentName эквивалентен типу string. Каждый новый компонент, помещаемый на форму, получает имя по умолчанию, автоматически образуемое путем добавления к названию компонента его номера в порядке помещения на 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 образуется из назначения компонента и его названия. Другим вариантом является указание в имени вместо названия компонента его префикса. Префикс является сокращением названия, например, для однострочного редактора Edit префикс может быть edt, для надписи Label — lЫ, для формы Form — fm. То есть для однострочного редактора, предназначенного для ввода фамилии сотрудника, подходящими именами будут NameEdit или edtName. Оба способа являются одинаково допустимыми, и на практике каждый разработчик использует тот, который для него наиболее удобен, или некоторый другой способ.

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

Свойства, связанные с размерами и положением компонента (например, Left и width), автоматически изменяют свои значения при перемещении компонента мышью и смене его размеров.

Если на форме выделено несколько компонентов, то в Инспекторе объектов доступны свойства, общие для всех этих компонентов. При этом сделанные в Инспекторе объектов изменения действуют для всех выделенных компонентов.

Отображаемое в Инспекторе объектов свойство может быть:

- простым (текстовым) — 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;
- перечислимым — значение свойства выбирается из раскрывающегося списка. Список раскрывается щелчком на стрелке, которая появляется при установке курсора в области значения свойства. Можно не выбирать, а ввести с помощью клавиатуры нужное значение, однако на практике это обычно не делается, так как ввести можно одно из предлагаемых значений. Кроме того, возрастает трудоемкость ввода значения и увеличивается вероятность ошибки. Используется для таких свойств, как FormStyle, Visible И ModalResult;

- множественным — значение свойства представляет собой комбинацию значений из предлагаемого множества. В Инспекторе объектов слева от названия свойства множественного типа содержится знак +. Формирование значения свойства выполняется с помощью дополнительного списка, раскрываемого двойным щелчком на названии 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;
- объектом — 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 названия может содержатся знак +, в этом случае управление его под свойствами выполняется как и для свойства множественного типа через раскрывающийся список. Этот список в левой части содержит названия 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 на эту кнопку. 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 операторов присваивания, к примеру, в обработчике события создания формы. Например, изменение заголовка кнопки Buttoni можно выполнить следующим образом:

```
Button1.Caption := 'Закрыть';
```

Однако это требует большего объема работ, чем в случае использования Инспектора объектов, кроме того, такие 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 нами устанавливаются с помощью операторов присваивания, а не через Инспектор объектов.

Отметим, что существуют свойства времени выполнения, недоступные через Инспектор объектов и 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.

1.3.3. 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 и при ее разработке. Форму можно перемещать по экрану, изменять ее размеры, минимизировать и максимизировать, а также закрыть 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. На форме, как правило, размещены компоненты, образующие интерфейс приложения, и разработчик должен для этих компонентов определить нужную реакцию на те или иные действия пользователя, например, на нажатие кнопки или включение переключателя. Эта реакция и определяет *функциональность* приложения.

Допустим, что при создании интерфейса приложения на форме была размещена кнопка Button, предназначенная для закрытия окна. По умолчанию эта кнопка получила имя и заголовок `Button1`, однако заголовок (свойство `Caption`) через Инспектор объектов был изменен на более осмысленный — закрыть. При выполнении приложения кнопку Buttoni можно нажимать с помощью мыши или клавиатуры. Кнопка отображает нажатие на себя 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 на нажатие кнопки.

Чтобы кнопка могла реагировать на какое-либо событие, для него необходимо создать или указать процедуру обработки события, которая будет вызываться при возникновении этого события. Для создания процедуры обработки события, или обработчика, прежде всего нужно выделить на форме кнопку и перейти на страницу событий Инспектора объектов, где указываются все возможные для кнопки события (рис. 1.6).

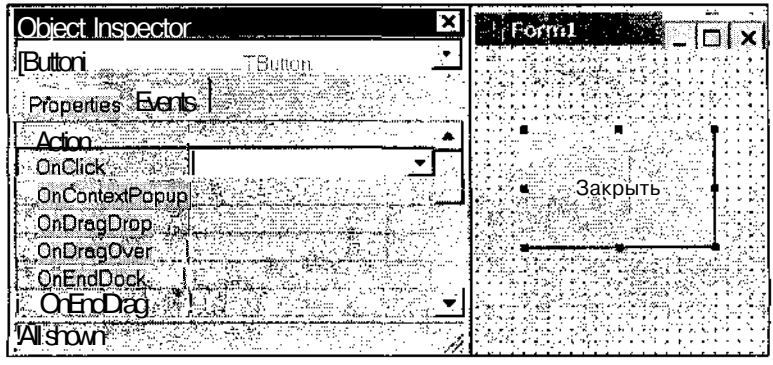


Рис. 1.6. Доступ к событиям компонента

Так как при нажатии на кнопку возникает событие Onclick, следует создать обработчик именно этого события. Для этого выполняется 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 на кнопку `Button1`. Так как кнопка должна закрывать окно, то в этом месте можно указать `Form1.Close` или просто close. При этом 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 курсивом нами выделен код, который набран программистом, все остальное было создано Delphi автоматически, в том числе включение заголовка процедуры-обработчика в описание класса формы Formi.

При создании обработчика события Delphi автоматически вносит изменения в файл модуля и файл описания. В описание класса формы добавляется строка

```
procedure Button1Click(Sender: TObject);
```

В тело модуля автоматически добавляется процедура обработки события, не содержащая функциональности.

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

end;
```

В дальнейших примерах описатели private и public и комментарии к ним в описании класса формы для краткости не приводятся.

Среда Delphi обеспечивает автоматизацию набора кода при вызове свойств и методов объектов и записи стандартных конструкций языка Object Pascal. 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. Название 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 условного оператора в Редактор кода будет автоматически добавлен следующий текст:

```
if  then
else
```

Название обработчика TForm1.Button1Click образуется прибавлением к имени компонента названия события без префикса On. Это название является значением события, для которого создан обработчик, в нашем случае — для события нажатия кнопки onclick. При изменении через Инспектор 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 вносил 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 в Инспекторе объектов щелчком на стрелке в области значения свойства раскрывается 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 анализ, для какого именно компонента или события вызвана процедура и в зависимости от этого выполнить нужные действия.

1.4. Средства интегрированной среды разработки

Интегрированная среда разработки в своем составе имеет много различных средств, служащих для удобной и эффективной разработки приложений. В этом подразделе рассматриваются наиболее общие элементы интегрированной среды разработки Delphi.

1.4.1. Встроенный отладчик

Интегрированная среда разработки включает встроенный отладчик приложений, в значительной степени облегчающий поиск и устранение ошибок в приложениях. Средства отладчика доступны через команды пункта меню **Run** (Выполнение) и подменю **View | Debug Windows** (Просмотр | Окна отладки) и позволяют выполнять такие действия, как:

- выполнение до указанного оператора (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) (рис. 1.7), вызываемом одноименной командой пункта меню Tools (Средства).

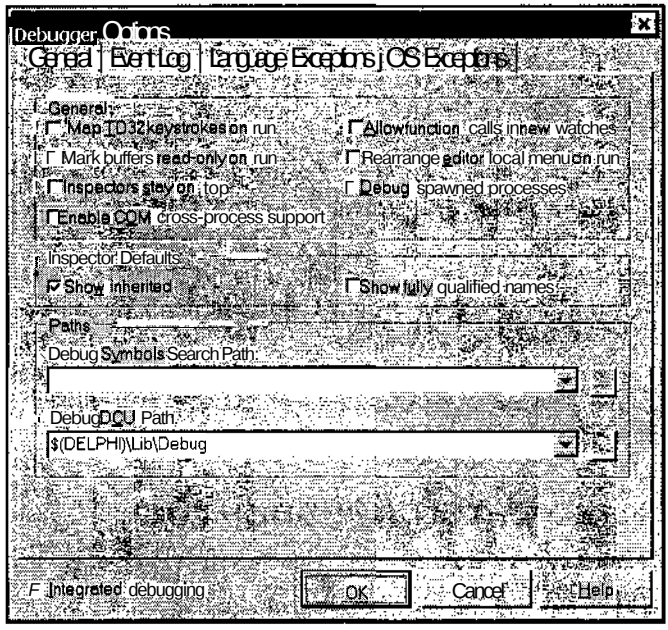


Рис. 1.7. Параметры отладчика

Включением/выключением отладчика управляет переключатель Integrated debugging (Интегрированная отладка), который по умолчанию включен, и отладчик автоматически подключается к каждому приложению. В ряде случаев, например, при отладке обработчиков исключительных ситуаций и проверке собственных средств обработки ошибок, этот переключатель отключают.

1.4.2. Обозреватель проекта

Обозреватель проекта (Project Browser или Browser) отображает список модулей, классов, типов, свойств, методов, 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 (Просмотр | Обозреватель).

1.4.3. Хранилище объектов

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

Вставить в приложение новый объект можно, вызвав командой **File | New | Other** (Файл | Новый | Другой) окно **New Items** (Новые элементы) выбора нового объекта из хранилища (рис. 1.8). Это окно можно также вызвать с помощью кнопки New (Новый) панели инструментов Менеджера проектов. В Хранилище находятся самые различные объекты, например, шаблоны приложений, форм, отчетов, а также Мастера форм.

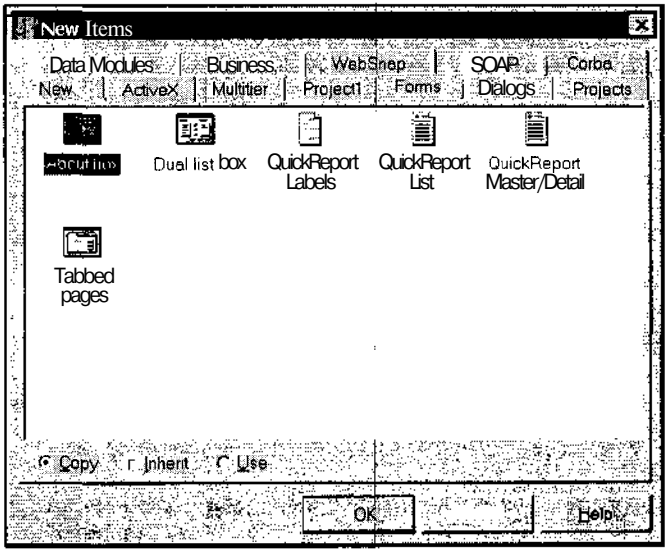


Рис. 1.8. Окно выбора нового объекта из хранилища

Все объекты объединены в группы, размещенные на отдельных страницах, к важнейшим из которых можно отнести следующие:

- New — базовые объекты;
- **Projectl — формы** создаваемого **приложения;**
- **Forms** — формы;
- **Projects** — проекты;
- **Data Modules** — модули данных;
- **Business** — Мастера форм.

Название страницы **Projectl** совпадает с названием создаваемого проекта, а сама страница в качестве шаблонов содержит 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 выбрать нужную страницу, после чего указать объект. На рис. 1.8 выбран объект **About box** (Информационное окно), расположенный на странице **Forms** (Формы). При нажатии **OK** происходит добавление объекта. Объекты можно добавлять к проекту различными способами, зависящими от состояния переключателей в нижней части окна выбора нового объекта. Обычно выбирается переключатель **Copy** (Копировать), при этом 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.

Объекты приложения, формы, фрейма, модуля данных и модуля кода также можно добавить к проекту через меню **File | New,** в котором содержатся команды добавления К Проекту Объектов Application, Form, Frame, Data Module И Unit.

1.4.4. Справочная система

Справочная система Delphi в свой состав включает: стандартную систему справки, справочную помощь через Internet 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, **Help Topics: Delphi Help** (Справочная система Delphi).

Доступ к справочной помощи через Internet выполняется с помощью команд меню **Help** (Помощь), которые вызывают запуск обозревателя (браузера), например Micrisoft Internet Explorer, с открытием соответствующей Web-страницы. Так, командой **Borland Home Page** (Домашняя страница Borland) осуществляется доступ к сайту компании Borland.

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

Глава 2

Язык Object Pascal

Язык Object Pascal является языком профаммирования Delphi и представляет собой объектно-ориентированное расширение стандартного языка Pascal. Система Delphi обеспечивает возможность визуального профаммирования на нем с помощью библиотеки визуальных компонентов VCL.

2.1. Основные понятия

2.1.1. Алфавит

Алфавит языка Object Pascal включает в себя следующие символы:

- 53 буквы — прописные (A—Z) и строчные (a—z) буквы латинского алфавита и знак подчеркивания (_);
- цифры — 0, 1,..., 9;
- 23 специальных символа — + - * / . , : ; = > < ' () {} [ ] # $ ^ @ и символ пробела.

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].

2.1.2. 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;
- идентификаторы пользователя.

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 служат для обозначения следующих заранее определенных разработчиками конструкций языка:

- типов данных;
- констант;
- процедур и функций.

В отличие от ключевых слов любой из стандартных идентификаторов можно переопределить. Так как это может привести к ошибкам, то стандартные идентификаторы лучше использовать без каких-либо изменений. Примерами стандартных идентификаторов являются слова sin, `Pi`, `Real`.

Идентификаторы пользователя применяются 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. Так как нельзя использовать специальные символы, то для наглядности отдельные составляющие идентификатора полезно выделять прописными буквами, например, btnOpen, или разделять их с помощью знака подчеркивания, который также относится к буквам, например, `Picture_ID`.
- Между двумя идентификаторами в программе должен быть по крайней мере один разделитель.

2.1.3. Структура программы

Исходный текст программы представляется в виде последовательности строк, в которой строка 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, в *исполнительной части* 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 указывается точка с запятой.

Структуру программы в общем случае можно представить следующим образом:

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

Пример. Подключение модулей.

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

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

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

Пример. Объявление меток.

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

В разделе *объявления констант* производится присваивание идентификаторам констант их постоянных значений. 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> = <Выражение>;
            ...
      <ИдентификаторN> = <Выражение>;
```

Пример. Объявление констант.

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

Тип констаты распознается компилятором автоматически на основании типа выражения.

В Delphi имеется большое количество констант, которые можно использовать без их предварительного объявления, например, Nil, True и MaxInt.

В разделе *описания типов* описываются типы данных пользователя. 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. Каждое имя типа и его описание отделяется точкой с запятой. Формат раздела:

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

Пример. Описание типов.

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

В Delphi имеется много *стандартных* типов, не требующих предварительного описания, например, Real, Integer, Char ИЛИ Boolean.

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. Формат раздела:

```
Var <Идентификаторы> : <Тип>;
      ...
    <Идентификаторы> : <Тип>;
```

Пример. Объявление переменных.

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

Замечание

Объявление переменных обеспечивает выделение памяти для размещения переменных в соответствии с их типом, но не присваивание им начальных значений. Программист должен самостоятельно задать нужные значения переменным перед тем, как использовать значения этих переменных.

Подпрограммой называют логически законченную и специальным образом оформленную часть программы, которая по ее имени может вызываться для выполнения из других точек программы неограниченное число раз. Подпрограммы в Паскале разделяются 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 ниже.

Раздел *операторов* начинается с ключевого слова Begin, после которого следуют операторы языка, разделенные точкой с запятой. Завершает этот раздел ключевое слово End, после которого указывается точка. Формат раздела:

```
Begin
  <Оператор1>;
  ...
  <ОператорN>;
End.
```

Здесь могут использоваться любые операторы языка, например, оператор присваивания или условный оператор.

2.1.4. 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 данными символами, может занимать несколько строк. Однострочный комментарий в начале строки содержит двойной слэш //.

Пример. Варианты комментариев.

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

Комментарий игнорируется компилятором и не оказывает никакого влияния на выполнение программы. С помощью комментариев на период отладки можно исключить какие-либо операторы программы.

2.1.5. Виды 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, свои значения они получают в процессе выполнения программы, причем эти значения можно изменять. К константам и переменным можно обращаться по именам. *Литерал* не имеет имени 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 констант и литералов распознается компилятором автоматически по указанному значению.

2.1.6. Типы данных

Тип определяет множество значений, которые могут принимать элементы программы, и совокупность операций, допустимых над этими значениями. Например, значения -34 и 67 относятся к целочисленному типу и их можно

умножать, складывать, делить и выполнять другие арифметические операции, а значения abcd и sdfh123 относятся к строковому типу и их можно сцеплять (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 включают в свой состав 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 тип, так как целое число автоматически преобразовывается в вещественное, но не наоборот.

2.1.7. Операторы

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

- простые;
- структурированные.

Например, к простым операторам относится оператор присваивания, к структурированным операторам — операторы разветвлений и циклов.

Правила записи операторов

Операторы разделяются точкой с запятой. Точка с запятой является разделителем операторов, и ее отсутствие между операторами является ошибкой. Наличие между операторами нескольких точек с запятой не является ошибкой, так как они обозначают пустые операторы. Отметим, что лишняя точка с запятой в разделе описаний и объявлений является синтаксической ошибкой.

Точка с запятой может не ставиться после слова begin и перед словом end, так как они являются операторными скобками, а не операторами. В условных операторах и операторах выбора точка с запятой не ставится после сло-

ва then и перед словом else. Отметим, что в операторе цикла с параметром наличие точки с запятой сразу после слова do синтаксической ошибкой не является, но в этом случае тело цикла будет содержать только пустой оператор.

2.2. Простые типы данных

Простые типы не содержат в себе других типов, и данные этих типов могут одновременно содержать одно значение. К простым относятся следующие типы:

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

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

2.2.1. Целочисленные типы

Целочисленные типы включают целые числа. Наиболее часто используется тип integer, допускающий значения в диапазоне от -2 147 483 648 до 2 147 483 647.

Для записи целых чисел можно использовать цифры и знаки + и -, если знак числа отсутствует, то число считается положительным. При этом число может быть представлено как в десятичной, так и в шестнадцатеричной системе счисления. Если число записано в шестнадцатеричной системе, то перед ним ставится знак $ (без пробела), а допустимый диапазон значений — `ОТ $00000000 ДО $FFFFFFFF`.

2.2.2. Литерные типы

Значениями *литерного типа* являются элементы из набора литер, то есть отдельные символы. В Object Pascal определен литерный тип `Char`, который занимает один байт, а для кодирования символов используется код американского национального института стандартов ANSI (American National Standards Institute).

К символам применимы следующие функции:

- chr(x) : char — возвращает символ с кодом, равным значению целочисленного выражения x;
- `UpCase` (C) : char — преобразует символ с к верхнему регистру.

2.2.3. Логический тип

В Object Pascal к логическому относится тип Boolean. Этот тип представлен двумя возможными значениями: True (истина) и False (ложь). Для представления логического значения требуется один байт памяти.

2.2.4. Интервальные типы

Интервальные типы описываются путем задания двух констант, определяющих границы допустимых для данных типов значений. Эти границы и определяют интервал (диапазон) значений. Компилятор для каждой операции с переменной интервального типа, если это возможно, проверяет, находится ли значение переменной внутри установленного для нее интервала, и в случае его выхода за границы выдает сообщение об ошибке. Во время выполнения программы при выходе значения интервального типа за границы интервала сообщение об ошибке не выдается, однако значение переменной будет неверным.

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

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

Пример. Описание переменных интервальных типов.

```
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.

2.2.5. Вещественные типы

Вещественные (действительные) типы включают в себя вещественные числа. Наиболее часто используется тип Real, допускающий максимальное значение $1{,}7 \times Ю^{308}$ и обеспечивающий точность 15—16 цифр мантиссы.

Запись вещественных чисел возможна в форме с фиксированной и в форме с плавающей точкой. Вещественные числа с фиксированной точкой записываются по обычным правилам арифметики. 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 знаком E (ИЛИ e). Примерами вещественных чисел являются `12.5`, `-137.0`, `+10E+3`.

Типы `Comp` и `currency` представляют вещественные числа с фиксированной точкой и введены для точных расчетов денежных сумм.

К выражениям вещественных типов применимы следующие функции:

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

2.3. Структурные типы данных

Структурные типы имеют в своей основе один или более других типов, в том числе и структурных. К структурным типам относятся:

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

Далее рассмотрим эти типы, кроме классов, которые будут изучены после подпрограмм и модулей.

2.3.1. Строки

Строки обеспечивает тип `string`, который представляет строку с максимальной длиной около $2x10^{31}$ символов. Символы строки кодируются в коде ANSI.

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

2.3.2. 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 номер (номера) элемента, заключенный в квадратные скобки, например, `arr1[3, 35]`, `arr1[3][35]` или `arr3[7]`.

Количество индексных позиций определяет мерность массива (одномерный, двумерный и т. д.), при этом мерность массива не ограничивается. В математике аналогом одномерного массива является вектор, а двумерного

массива — матрица. Индексы элементов массива должны принадлежать порядковому типу. Разные индексы одного и того же массива могут иметь различные типы. Наиболее часто типом индекса является целочисленный тип.

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

```
Array [Тип индексов] of <Тип элементов>;
```

Пример. Объявление статических массивов.

```
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 юоо элементов — ю строк и юо 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 соответственно на 81 символ и 26 целых чисел.

2.3.3. Множества

Множество представляет собой совокупность элементов, выбранных из предопределенного 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 [ ].

В Delphi множественные типы используются, например, для описания типа кнопок в заголовке окна TBorderIcons или типа 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.

2.4. 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. В качестве операндов могут использоваться константы (литералы), переменные, элементы массивов и обращения к функциям.

Операции определяют действия, которые выполняются 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 знак ее 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.*

2.4.1. 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, результатом которых является число.

Тип значения арифметического выражения определяется типом операндов и 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.

Существуют бинарные операции (+ — сложение, — вычитание, * — умножение и / — деление), которые применяются к двум операндам, и унарные операции (+ — сохранение знака и — — отрицание знака), относящиеся к одному операнду.

Унарные арифметические операции относятся к знаку числа и не меняют типа числа.

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

- Abs (X) — абсолютное значение x;
- Sqrt(X) — квадратный корень 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.

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

Пример. Арифметические выражения.

```
(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.

Пример. Использование арифметических операций для целочисленных операндов.

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

Замечание

В Object Pascal отсутствует операция возведения в степень. Возведение числа (выражения) в целую степень можно выполнить в цикле путем многократного умножения на данное число. Возведение положительного не нулевого числа X в любую степень A можно выполнить с помощью выражения `Exp(A * Ln(X))`.

2.4.2. Логические выражения

Результатом *логического выражения* является логическое значение True или False. Логические выражения чаще всего используются в условном операторе и в операторах 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,
- o — не равно.

Операции сравнения выполняются после вычисления соответствующих выражений. Результатом операции сравнения является значение False, если соответствующее отношение не имеет место, и значение True, если соответствующее отношение имеет место.

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 (типа boolean) (табл. 2.1) при применении их к логическим выражениям (операндам логического типа) вырабатывают значения логического типа (boolean). Логические операции And, `Or` и `Xor` являются бинарными, операция Not — унарной. Отметим, что в Object Pascal есть од-

ноименные логические (поразрядные) операции, выполняющие поразрядные действия над битами (разрядами) целых чисел.

Таблица 2.1. *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

Пример. Логические выражения.

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

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

2.4.3. Строковые выражения

Результатом строкового выражения является строка символов. Для строк можно применять операцию +, выполняющую соединение (конкатенацию) строк, а 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;

- 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 числа зависит от представления числа в строке. Параметр code возвращает код результата операции, если операция выполнена успешно, то возвращается значение ноль;
- str(x [: width [: Decimals]]; var s) — преобразование значения численного выражения x в строку s.

Кроме отмеченных подпрограмм, большое количество процедур и функций для работы со строкам содержится в модуле SysUtils. Выделим 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;
- 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 — преобразование символов строки s к верхнему регистру;
- LowerCase (const S: String): String — преобразование символов строки s к нижнему регистру;

- Trim(const s: string) : string — удаление из начала и конца строки s пробелов и управляющих символов;
- TrimLeft(const S: String): String — удаление пробелов И управЛЯЮщих символов из начала 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 О И Timed, возвращающие текущие значения даты и времени.

Пример. Строковые выражения.

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

Переменная s должна принадлежать строковому типу, а x — к вещественному.

2.5. Простые операторы

Простыми называются операторы, не содержащие в себе других операторов. К ним относятся:

- оператор присваивания;
- оператор перехода;
- пустой оператор;
- оператор вызова процедуры.

2.5.1. Оператор присваивания

Оператор присваивания является основным оператором языка. Он предписывает вычислить выражение, заданное в его правой части, и присвоить результат переменной, имя которой расположено в левой части оператора. Переменная и выражение должны иметь совместимый тип, например, вещественный и целочисленный, но не наоборот. Допустимо присваивание любого типа данных, кроме файлового. Формат оператора 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 сначала вычисляется значение выражения и потом оно присваивается указанной

переменной. Поэтому при условии, что x является числовой переменной и имеет определенное значение, будет допустимой и правильной следующая конструкция:

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

Пример. Операторы присваивания.

```
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;
```

2.5.2. Оператор перехода

Оператор перехода предназначен для изменения естественного порядка выполнения операторов программы. Он используется в случаях, когда после выполнения некоторого оператора требуется выполнить не следующий по порядку, а какой-либо другой помеченный меткой оператор. Метка, стоящая перед оператором, отделяется от него двоеточием.

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

```
goto <Метка>;
```

Пример. Использование оператора перехода.

```
Label ml;
...

goto ml;
...

ml: <Оператор>;
```

Передавать управление с помощью оператора перехода можно на операторы, расположенные в тексте программы выше или ниже оператора перехода. Запрещается передавать управление операторам, находящимся внутри структурированных операторов, а также операторам, находящимся в других блоках (процедурах, функциях).

2.5.3. Пустой оператор

Пустой оператор представляет собой точку с запятой и может быть расположен в любом месте программы, где допускается наличие оператора. Как и другие операторы, пустой оператор может быть помечен меткой. Пустой оператор не выполняет никаких действий и может быть использован для передачи управления в конец цикла или составного оператора.

2.5.4. Оператор вызова процедуры

Оператор вызова процедуры служит для активизации стандартной или предварительно описанной пользователем процедуры и представляет собой имя этой процедуры со списком передаваемых ей параметров. Более подробно этот оператор будет рассмотрен при изучении процедур.

2.6. Структурированные операторы

Структурированные операторы представляют собой конструкции, построенные по определенным правилам из других операторов. К структурированным операторам относятся:

- составной оператор;
- условный оператор;
- операторы выбора;
- операторы цикла (повтора);
- оператор доступа.

2.6.1. Составной оператор

Составной оператор представляет собой группу из произвольного числа любых операторов, отделенных друг от друга точкой с запятой, и ограниченную операторными скобками begin и end. Формат составного оператора:

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

Независимо от числа входящих в него операторов, составной оператор воспринимается как единое целое и может располагаться в любом месте программы, где допускается наличие оператора. Наиболее часто составной оператор используется в условных операторах и операторах цикла.

Пример. Составной оператор.

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

Приведенный составной оператор может быть использован в условном операторе при проверке выполнимости некоторого условия, скажем, для указания действий при возникновении ошибки.
Составные операторы могут вкладываться друг в друга.

2.6.2. Условный оператор

Условный оператор обеспечивает выполнение или невыполнение некоторых операторов в зависимости от соблюдения определенных условий. Условный оператор в общем случае предназначен для организации разветвления программы на два направления и имеет формат:

```
if <Условие> then <Оператор1> [ else <0ператор2> ];
```

Условие представляет собой выражение логического типа. Оператор работает следующим образом: если условие истинно (имеет значение True), то выполняется Оператор1, в другом случае выполняется 0ператор2. Оба оператора могут быть составными.

Условный оператор может быть записан в сокращенной форме, когда слово else и оператор после него отсутствуют. В этом случае при невыполнении условия не выполняется никакой оператор.

Для организации разветвлений на три направления и более можно использовать несколько условных операторов, вложенных друг в друга. При этом каждое else соответствует тому then, которое непосредственно ему предшествует. Из-за возможных ошибок следует избегать большой вложенности условных операторов друг в друга.

Пример. Условные операторы.

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

2.6.3. Оператор выбора

Оператор выбора является обобщением условного оператора и позволяет сделать выбор из произвольного числа имеющихся вариантов, то есть организовать разветвления на произвольное число направлений. Этот оператор состоит из выражения, называемого *селектором,* списка вариантов и необязательной ветви else, имеющей тот же смысл, что и в условном операторе. Формат оператора выбора:

```
case <Выражение-селектор> of
       <Список1> : <Оператор1>;
         ...
       <СписокN> : <ОператорN>
       else <Оператор>;
end;
```

Выражение-селектор должно быть порядкового типа. Каждый вариант представляет собой список констант, отделенных двоеточием от относящегося к данному варианту оператора, возможно, составного. Список констант выбора состоит из произвольного количества значений и диапазонов, отделенных друг от друга запятыми. Границы диапазона записываются двумя константами через разделитель ".." . Тип констант должен совпадать с типом выражения-селектора.

Оператор выбора выполняется следующим образом:

- вычисляется значение выражения селектора;
- производится последовательный просмотр вариантов на предмет совпадения значения селектора с константами и значениями из диапазонов соответствующего списка;
- если для очередного варианта этот поиск успешный, то выполняется оператор этого варианта. После этого выполнение оператора выбора заканчивается;
- если все проверки оказались безуспешными, то выполняется оператор, стоящий после слова else (при его наличии).

Пример. Оператор 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.

2.6.4. Операторы цикла

Операторы цикла используются для организации циклов (повторов). *Цикл* представляет собой последовательность операторов, которая может выполняться более одного раза. Группу повторяемых операторов называют телом цикла. Для построения цикла в принципе можно использовать ранее рассмотренные условный оператор и оператор перехода. Однако в большинстве случаев удобно использовать операторы цикла. Всего имеется три вида операторов цикла:

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

Обычно, если количество повторов известно заранее, то применяется оператор цикла с параметром, в противном случае — операторы с пост- или предусловием (чаще используются оператор с предусловием).

Выполнение оператора цикла любого вида может быть прервано с помощью оператора перехода goto или предназначенной для этих целей процедуры без параметров Break, которая передает управление на следующий за оператором цикла оператор.

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

Операторы циклов могут быть вложенными друг в друга.

Оператор цикла с параметром

Оператор цикла с параметром имеет два следующих формата:

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

И

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

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

Оператор цикла обеспечивает выполнение тела цикла, которым является оператор после слова do, до полного перебора всех значений параметра цикла от начального до конечного с соответствующим шагом. Шаг параметра всегда равен 1 для первого формата цикла и -1 — для второго формата. То есть 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.

Пример. Циклы с параметром.

```
var n, k: integer;
...
s := 0;
for n := 1 to 10 do s := s + m[n];
for к := 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 из десяти значений массива т. Во втором случае два цикла вложены один в другой, и в них пересчитываются значения элементов двумерных массивов arrl и arr2.

Оператор цикла с предусловием

Оператор цикла с предусловием целесообразно использовать в случаях, когда число повторений тела цикла заранее неизвестно и тело цикла может не выполняться. Во многом этот оператор аналогичен оператору repeat...until, но проверка условия выполняется в начале оператора. Формат оператора цикла с предусловием:

```
while <Условие> do <Оператор>;
```

Оператор тела цикла выполняется до тех пор, пока логическое выражение не примет значение False, то есть в отличие от цикла с постусловием, цикл выполняется при значении логического выражения True.

Пример. Оператор цикла с предусловием.

Рассмотрим расчет суммы из десяти значений массива т.

```
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 выполняется (значение логического выражения равно False), то тело цикла не выполняется ни разу, и происходит переход на следующий за оператором цикла оператор.

2.6.5. Оператор доступа

Оператор 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. Оператор доступа имеет следующий основной формат:

with <Имя объекта> do <Оператор>

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

Пример. Использование оператора доступа.

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

или

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

В обоих приведенных примерах на форме красным пером толщиной пять пикселов рисуется прямоугольник. Для обращения к свойствам и методу (процедуре) поверхности рисования формы удобно использовать оператор доступа (второй вариант).

Для наглядности в примерах этой книги оператор доступа используется редко, а составные имена обычно указываются полностью, как в первом варианте приведенного примера.

2.7. Подпрограммы

Подпрограмма представляет собой группу операторов, 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 имеет два этапа:

- описание подпрограммы;
- вызов подпрограммы.

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

В различных модулях Delphi имеется большое число стандартных подпрограмм, которые можно вызывать без предварительного описания. Некоторые из них приведены при описании типов данных и выражений. Кроме того, программист может создавать свои подпрограммы, которые также называют *пользовательскими*.

Подпрограммы делятся на *процедуры* и *функции,* которые имеют между собой много общего. Основное различие между ними заключается в том, что функция может возвращать под своим именем в качестве результата значение и соответственно может использоваться в качестве операнда 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 используется последний способ. При этом различают параметры и аргументы. *Параметры* (формальные параметры) являются элементами подпрограммы и используются при описании алгоритма, выполняемого подпрограммой.

Аргументы (фактические параметры) являются элементами вызывающей программы. Они замещают параметры при вызове подпрограммы. При этом осуществляется 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 операторов подпрограммы и возвращает управление вызывающей программе.

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

2.7.1. Процедуры

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

Procedure <Имя> [(формальные параметры)];

Для обращения к процедуре используется *оператор вызова* процедуры. Он включает имя процедуры и список аргументов, заключенный в круглые скобки.

Пример. Использование процедур.

Рассмотрим процедуру обработки события нажатия кнопки Buttoni, в которой 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;
```

Процедура DecodeDate выполняет декодирование даты на отдельные составляющие (год, месяц и день) и может быть использована без предварительного описания, так как она 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`. Это описание можно вынести за пределы обработчика, в этом случае процедуру `ChangeStr` можно будет вызывать не только из данного обработчика.

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

2.7.2. Функции

Описание *функции* состоит из заголовка и блока. *Заголовок* включает ключевое слово Function, имя функции, необязательный список формальных параметров, заключенный в круглые скобки, и тип возвращаемого функцией значения. Заголовок имеет формат:

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

Возвращаемое значение может иметь любой тип, кроме файлового.

Блок функции представляет собой локальный блок, по структуре аналогичный блоку процедуры. В теле функции должен быть хотя бы один оператор присваивания, в левой части которого стоит имя функции. Именно он и определяет значение, возвращаемое функцией. Если таких операторов несколько, то результатом функции будет значение последнего выполненного оператора присваивания. В этих операторах вместо имени функции можно 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 переменной Result можно в любой момент получить внутри блока доступ к текущему значению функции.

Замечание

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

Вызов функции осуществляется по ее имени с указанием в круглых скобках списка аргументов, которого может и не быть. При этом аргументы должны попарно соответствовать параметрам, указанным в заголовке функции, и иметь те же типы. В отличие от процедуры, имя функции может входить как операнд в выражения.

Пример. Использование функций.

Рассмотрим процедуру обработки события нажатия кнопки Button1, в которой 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 предыдущего примера.

Вызов функции используется в выражении оператора присваивания.

2.7.3. Параметры и аргументы

Параметры являются элементами подпрограммы и используются при описании ее алгоритма. Аргументы указываются при вызове подпрограммы и замещают параметры при выполнении подпрограммы. Параметры могут

иметь любой тип, включая структурированный. Существует несколько видов параметров:

- значение;
- константа;
- переменная;
- нетипизированная константа и переменная.

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

```
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 выполнении подпрограммы были подставлены вместо фактических параметров-значений.

Группа параметров, перед которыми в заголовке подпрограммы стоит слово const и за которыми следует их тип, называются *параметрами-константами*. Например, в следующем 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.

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

```
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 образом можно вернуть результаты из подпрограммы по окончании ее работы.

Для параметров-констант и параметров-переменных можно не указывать их тип, в этом случае они считаются *нетипизированными*. В этом случае подставляемые на их место аргументы могут быть любого типа, и программист должен самостоятельно интерпретировать типы параметров в теле подпро-

граммы. Примером задания нетипизированных параметров-констант и параметров-переменных является следующее описание:

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

Отметим, что группы параметров в описании подпрограммы разделяются точкой с запятой.

2.8. Особенности объектно-ориентированного программирования

Язык Object Pascal является объектно-ориентированным расширением языка Pascal и реализует концепцию объектно-ориентированного программирования. Это означает, что создаваемое приложение состоит из объектов, которые взаимодействуют между собой. Каждый объект имеет свои свойства, то есть характеристики (атрибуты) этого объекта, методы, определяющие поведение этого объекта, и события, на которые реагирует объект.

2.8.1. Классы и объекты

В языке Object Pascal классы являются специальными типами данных и используются для описания объектов. Соответственно *объект,* имеющий тип какого-либо класса, является *экземпляром* этого класса или переменной этого типа.

Класс представляет собой особый тип записи, имеющий в своем составе такие элементы (члены), 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 оператора присваивания; с другой стороны, внутри класса доступ к значениям свойств выполняется методами 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. Описание 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
  CCirclel: 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. Обращение к полям объекта.

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

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

2.8.2. Поля

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

Пример. Описание полей.

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

Здесь новый класс TNewCiass 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 удаление или переопределение этих полей невозможно.

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

2.8.3. Свойства

Свойства реализуют механизм доступа к полям. Каждому свойству соответствует 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 класса, с помощью которых выполняется чтение значения поля, связанного со свойством, или запись нового значения в это поле.

Пример. Описание 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 другим классам, используются свойства code, sign и Note, соответственно.

2.8.4. 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. 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 через параметры.

2.8.5. 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. Система Delphi преобразовывает сообщение в свой формат.

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.

Обычно в Delphi не возникает необходимость обработки непосредственных сообщений Windows, т. к. в распоряжение программиста предоставляются события, работать с которыми намного 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 называют *обработчиками событий*.

Пример. Назначение обработчика события.

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

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

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

```
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.

2.8.6. 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 имеет непосредственную связь с интегрированной средой разработки приложений Delphi. В VCL содержатся невизуальные и визуальные компоненты, а также другие классы, начиная с абстрактного класса TObject. При этом все компоненты являются классами, но не все классы являются компонентами.

Все классы VCL расположены на определенном уровне иерархии и образуют дерево (иерархию) классов. Знание происхождения объекта оказывает значительную помощь при его изучении, так как 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 независимого переключателя `CheckBox` соответственно. Фрагмент иерархии классов с важнейшими классами показан на рис. 2.1.

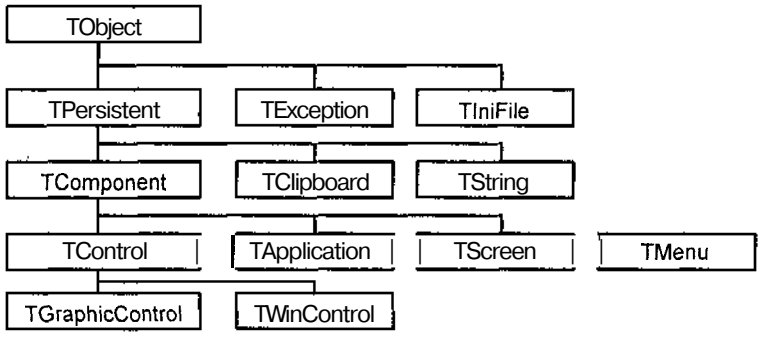


Рис. 2.1. Фрагмент иерархии классов

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

Глава 3

Визуальные компоненты

3.1. Страницы с визуальными компонентами

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

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

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

- Frames — фреймы;
- `MainMenu` — главное меню;
- Рорирмепи — всплывающее меню;
- Label — надпись;
- Edit — однострочный редактор;
- Memo — многострочный редактор;
- Button — стандартная кнопка;
- checkbox — независимый переключатель;

- RadioButton — зависимый 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

Компоненты перечислены последовательно в порядке расположения их пиктограмм на странице. Первая пиктограмма соответствует не компоненту, а стрелке отмены выбора компонента на странице.

На странице Additional (Дополнительная) (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 — графический образ;
- Shape — геометрическая фигура;
- Bevel — фаска;
- ScrollBox — область прокрутки;
- checkListBox — список независимых переключателей;
- 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;
- CustomizeDlg — диалог настройки.

Рис. 3.2. Страница **Additional**

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

- TabControl — закладка;
- PageControl — блокнот;
- ImageList — список графических образов;
- RichEdit — полнофункциональный тестовый редактор;
- TrackBar — ползунок;
- ProgressBar — индикатор хода работ;
- 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.

Приведенные страницы соответствуют 6-й версии Delphi.

Рис. 3.3. Страница Win32

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

3.2. Базовый класс *TControl*

В библиотеке визуальных компонентов VCL (Visual Component Library) для большинства визуальных компонентов базовым является класс TControl, производимый от класса `TComponent`. Класс TControl включает в себя общие для визуальных компонентов свойства, события и методы. 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, кнопка Button и поле редактирования Edit. Оконный элемент управления может получать фокус ввода, а также содержать другие компоненты, в том числе неоконные. 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 информации, получение фокуса ввода обычно отображают с помощью пунктирного черного прямоугольника. При этом, например, для кнопки Button этот прямоугольник появляется вокруг ее заголовка, а для списка ListBox прямоугольник выделяет выбранную в текущий момент времени строку. Выбранная строка может окрашиваться в какой-либо цвет, чаще всего синий.

Неоконные элементы управления не могут получать фокус ввода и быть родителями других интерфейсных элементов. Достоинством неоконных эле-

ментов управления по сравнению с оконными является меньшее расходование ресурсов. Неоконными элементами управления являются, например, компоненты Label и DBText. Для неоконных элементов управления базовым является класс `TGraphicControl`, производимый непосредственно от класса `TControl`.

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

3.3. Свойства

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

Рассмотрим наиболее общие свойства визуальных компонентов, описав свойства в алфавитном порядке. Отметим, что отдельные компоненты имеют не все рассматриваемые ниже свойства, например, редактор Edit не имеет свойства Caption, а надпись Label — свойства Readonly.

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 свойству Name.

Пример. Создание 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 получет имя edtName. Новому редактору устанавливаются текстовое значение и координаты его размещения в контейнере — владельце этого компонента. Владельцем нового редактора является форма Forml.

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.

Свойство Caption типа TCaption содержит строку для *надписи заголовка* компонента. Отметим, что тип TCaption равен типу string. Отдельные символы в заголовке могут быть подчеркнуты, они обозначают комбинации клавиш быстрого доступа: нажатие на клавишу 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&Звpат тары';          <Alt>+<3>
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 равен коду $00,

что соответствует отображению цвета, наиболее близкого к заданному 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 $оо, то соответствующий базовый цвет выключен. Отсутствие базовых цветов приводит к черному цвету, а их максимальная интенсивность образует белый цвет.

Например, черному цвету соответствует код $оооооо, 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.

Таблица 3.1. *Константы основных цветов*

Константа	Цвет	Значение
clAqua	Ярко-голубой	$FFFF00
clBlack	Черный	$000000
clBlue	Синий (голубой)	$FF0000
clFuchsia	Сиреневый	$FF00FF
clGray	Серый	$808080
clGreen	Зеленый	$008000
clLime	Ярко-зеленый	$00FF00
clMaroon	Темно-красный	$000080
clNavy	Темно-синий	$800000
clOlive	Оливковый	$008080
clPurple	Фиолетовый	$800080
clRed	Красный	$0000FF
clSilver	Серебряный	$C0C0C0
clTeal	Сине-зеленый	$808000
clWhite	Белый	$FFFFFF
clYellow	Желтый	$00FFFF

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.

Другой набор констант (табл. 3.2) указывает цвета в составе системной палитры Windows, установленные на вкладке **Appearance** (Появление) 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	Пункт 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 текста
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 инструментальных средств
clInfoBk	Фон инструментальных средств

Свойство cursor типа TCursor определяет *вид указателя мыши* при размещении его в области компонента. Delphi предлагает более двадцати предо-

пределенных видов указателя мыши и соответствующих им констант, основными из которых являются следующие:

- crDefault — указатель имеет вид по умолчанию (обычно стрелка);
- crNone — указатель не виден;
- crArrow — указатель имеет вид стрелки;
- crCross — указатель имеет вид креста;
- crDrag - указатель имеет вид стрелки с листом бумаги;
- crHourGlass — указатель имеет вид песочных часов.

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 (переместить и оставить), 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 dmManuai, то элемент управления можно перемещать мышью в любой момент. Кроме установки свойству DragMode требуемого значения, программист должен выполнить кодирование действий, которые управляют перемещением элемента, то есть подготовить обработчики событий, связанных с операцией перемещения.

Свойство Enabled типа Boolean определяет *активность* компонента, то есть 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 следующего оператора присваивания

```
Edit1.Text := 'Иванов П.А.';
```

Отметим, что запретить изменение значения компонента можно также, установив свойству Readonly значение True.

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

- Name типа TFontName определяет название шрифта, например, Arial или Times New Roman. Отметим, что свойство Name шрифта не связано с одноименным свойством самого компонента.
- size типа integer задает размер шрифта в пунктах. Пункт равен 1/72 дюйма.
- 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.
- style типа TFontStyle задает стиль шрифта и может принимать комбинации следующих значений:
  - fsItalic — курсив;
  - fsBold — полужирный;
  - fsUnderline — с подчеркиванием;
  - fsStrikeOut — с перечеркиванием.
- Color типа TColor управляет цветом текста.

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

Пример. Задание цвета компонента.

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

Здесь для редактора Editl устанавливаются зеленый цвет текста и синий цвет фона.

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 свойству `ShowHint` типа Boolean. По умолчанию showHint имеет значение False, и подсказки не отображаются.

Свойство PopupMenu типа TPopupMenu указывает *локальное всплывающее меню,* появляющееся при нажатии правой кнопки мыши и размещении указателя в области компонента. Чтобы ассоциированное с компонентом меню 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>, то есть 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 установить соответствующие значения свойству 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 осуществляет за этим контроль, 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) (рис. 3.4), при этом Delphi автоматически присваивает значения свойству

TabOrder этих компонентов. Вызов диалогового окна осуществляется одноименной командой меню **Edit** (Правка).

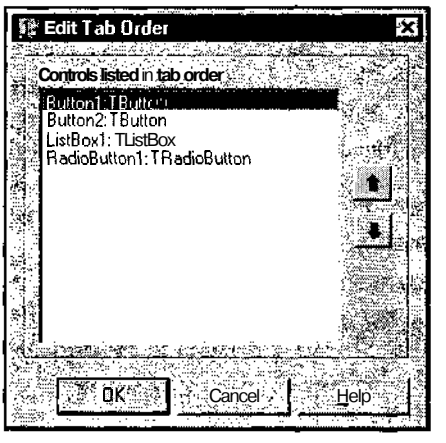


Рис. 3.4. Окно управления порядком табуляции

Свойство Readonly типа Boolean определяет, разрешено ли связанному с вводом и редактированием информации управляющему элементу 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, следующим образом:

```
Edit1.ReadOnly := true;
Edit1.Text := 'Новый текст';
```

Замечание

Даже если изменение информации 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 (TField) также имеет свойство Readonly, разрешающее или запрещающее изменение его значения. Если свойству установлено значение True, то программисту также запрещено изменять значение поля, и при попытке это сделать генерируется исключительная ситуация.

Визуальные компоненты для таких свойств, как Color, Ctl3D, Font и ShowHint, могут принимать значения соответствующих свойств родительского элемента управления, например, формы. Источники значения

для указанных свойств определяют следующие свойства-признаки 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 нужное значение у соответствующего свойства родителя. Например, если для формы изменить размер шрифта на значение 12, то шрифт будет изменен и для всех компонентов, расположенных на 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 свое собственное значение, а не родительское, и при необходимости наследования программист должен назначить признаку наследования значение True.

Свойство visible типа Boolean управляет видимостью компонента. Если ему 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.

Пример. Управление видимостью компонентов.

```
Edit1.Visible := true;
Edit2.Visible := false;
```

Здесь однострочный редактор Editl устанавливается видимым пользователю, при этом скрывается однострочный редактор Edit2.

3.4. События

Визуальные компоненты способны генерировать и обрабатывать достаточно большое число (несколько десятков) событий различных видов. К наиболее общим группам событий можно отнести следующие:

- выбор управляющего элемента;

- перемещение указателя мыши;
- вращение колеса мыши;
- нажатие клавиш клавиатуры;
- получение и потеря управляющим элементом фокуса ввода;
- перемещение объектов методом drag-and-drop.

В языке Object Pascal, который лежит в основе Delphi, события также являются свойствами и принадлежат к соответствующему типу. Большинство событий носят нотификационный (уведомляющий) характер и принадлежат типу TNotifyEvent, описанному следующим образом:

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

Как видно из описания, 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.

Рассмотрим наиболее часто используемые события.

При выборе управляющего элемента 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.

Пример. Процедура 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 на управляющий элемент, находящийся в фокусе ввода, например, для компонента Button — с помощью клавиш <Пробел> или <Enter>, а для компонента `CheckBox` — клавишей <Пробел>.

При щелчке любой кнопкой мыши генерируются еще два события: `OnMouseDown` типа TMouseEvent, возникающее при нажатии кнопки мыши, и `OnMouseUp` типа TMouseEvent, возникающее при отпускании кнопки.

При двойном щелчке левой кнопкой мыши в области компонента, кроме того, генерируется событие `OnDblClick` типа TNotifyEvent. События возни-

кают В следующем ПОрЯДКе: OnMouseDown, OnClick, OnMouseUp, OnDblClick, OnMouseDown, OnMouseUp.

При *перемещении указателя мыши* над визуальным компонентом непрерывно вырабатывается событие onMouseMove типа TMouseMoveEvent. Последний описан следующим образом:

```
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 указываются 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, ssctri]. Если не нажата ни одна клавиша, то параметр Shift принимает пустое значение [ ].

Пример. Отображение координат указателя мыши.

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
          X, Y: Integer);
begin
Form1.Caption := 'Координатыуказ. мыши: ' + IntToStr(X) + ' и ' + Int-
ToStr(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 управляющим элементом. Координаты x и Y отсчитываются в пикселах от левого верхнего угла формы, начиная с нуля.

При *работе с клавиатурой* генерируются события onKeyPress и OnKeyDown, возникающие при нажатии клавиши, а также событие OnKeyUp, возникаю-

щее 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 следующим образом:

```
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 ноль (#0), то это соответствует отмене нажатия клавиши.

Замечание

Обработчик события OnKeyPress не реагирует на нажатие управляющих клавиш, тем не менее, параметр Key содержит код символа с учетом регистра, который определяется состоянием клавиш <Caps Lock> и <Shift>.

Пример. 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 на любую клавишу. Тип TKeyEvent описан как

```
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 других. 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 o .

В обработчиках событий, связанных с нажатием клавиш, можно также обрабатывать комбинации управляющих и алфавитно-цифровых клавиш, например, <Alt>+<S>.

Пример. Обработка нажатий управляющих и алфавитно-цифровых клавиш.

```
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 на клавишу <Tab> не возникают события OnKeyPress И OnKeyUp.

При получении фокуса оконным элементом управления возникает событие onEnter типа TNotifyEvent. Оно генерируется при активизации управляющего элемента любым способом, например, щелчком мыши или с помощью клавиши <Tab>. В случае потери фокуса ввода оконным элементом управления возникает событие OnExit типа TNotifyEvent.

Пример. Процедуры обработки 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.

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.

3.5. Методы

С визуальными компонентами, как и с другими объектами, связано большое количество методов, позволяющих создавать и разрушать объекты, прорисовывать их, отображать и скрывать, а также выполнять другие операции. Здесь приведены несколько наиболее общих для всех компонентов методов.

Процедура `SetFocus` *устанавливает фокус ввода* на оконный элемент управления. Если элемент управления в данный момент времени не способен получать фокус ввода, то возникает ошибка. Поэтому в случае вероятного возникновения ошибки целесообразно предварительно выполнить соответствующую проверку. Проверить возможность активизации компонента позволяет функция `CanFocus: Boolean`, возвращающая значение True, если управляющий элемент 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.

Пример. Получение компонентом `Edit1` фокуса ввода.

```
If Edit1.CanFocus then Edit1.SetFocus;
```

Перед получением фокуса производится проверка на возможность передачи фокуса компоненту.

Метод `Clear` служит для *очистки содержимого компонентов,* которые могут содержать внутри себя текстовую информацию.

Пример. Очистка содержимого компонентов `ListBox1` и `Memo1`.

```
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.

Глава 4

Работа с текстом

При работе с информацией (текстом) требуется выполнять ее отображение, ввод и редактирование. Подчеркнем, что мы рассматриваем понятие текста в широком смысле, предполагая, что в состав текста могут входить буквы и цифры. При работе с текстовыми данными они могут объединяться в коллекции или массивы строк. В Delphi важную роль в операциях с такими данными играет класс Tstrings. Для отображения надписей (текста, используемого в качестве заголовков для некоторых управляющих элементов) служит компонент Label. Ввод и редактирование информации выполняется в специальных полях или областях формы. При необходимости пользователь может изменить отображаемые данные. Для обеспечения возможности редактирования информации Delphi предлагает различные компоненты, например, Edit, `MaskEdit`, Memo И RichEdit.

4.1. 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.

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 отличий не имеет.

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 так далее. 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 исключительной ситуации.

Свойство Count типа integer задает *число элементов* в списке. Поскольку первый элемент списка имеет нулевой номер, то номер последнего элемента равен Count-1.

Пример. Присваивание элементам списка ListBox1 новых значений.

```
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 на позицию с номером, определяемым параметром index. При этом элементы списка, находившиеся до операции вставки в указанной позиции и ниже, смещаются вниз.

Пример. Добавление к комбинированному списку ComboBox1 строки "Нажата кнопка Button1".

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ComboBox1.Items.Add('Нажата кнопка Button1');
end;
```

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. Оба метода позволяют при вызове увеличить содержимое списка более, чем на один элемент. Проверить, требуется ли операция согласования списков или нет, можно с помощью метода Equals.

Функция AddObject (const S: String; AObject: TObject): Integer *добавляет в конец списка* строку s и связанную с ней ссылку на объект, указываемую параметром AObject.

Процедура Assign (Source: TPersistent) *присваивает один объект* другому, при этом объекты должны иметь совместимые типы. Применительно к спискам в результате выполнения процедуры происходит копирование информации 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 одинаковую* текстовую информацию. 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.

Пример. 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, то содержимое списка ListBoxl копируется в список ListBox2, в результате чего содержимое списков становится одинаковым.

Для *удаления элементов списка* используются методы Delete и Clear. Процедура 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.

Пример. Удаление элемента списка.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  ComboBox1.Items.Delete(4);
end;
```

Здесь при нажатии кнопки Button2 из комбинированного списка ComboBoxl удаляется пятая строка.

Процедура clear *очищает список,* удаляя все его элементы.

Пример. Очистка элементов списка.

```
procedure TForm1.btnClearPersonalListClearClick(Sender: TObject);
```

```
begin
  lbPersonal.Items.Clear;
end;
```

При нажатии кнопки `btnClearPersonalList` происходит очистка списка lbPersonal.

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.

Класс TStrings имеет 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 списка строк из этого файла. Символы файла кодируются в системе **ANSI.**

Процедура SaveToFiie (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.

Пример. Сохранение строковых элементов списка в файле.

```
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 очищается. Если заданный файл отсутствует на диске, то возникает исключительная ситуация.

Пример. Загрузка содержимого списка.

```
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 редактора String List editor (Строковый редактор) — рис. 4.1. Его вызов выполняется через Инспектор объектов двойным щелчком мыши в области значения свойства типа `TStrings`.

Строковый редактор позволяет добавлять строки в список, удалять их из списка и изменять содержимое имеющихся строк.

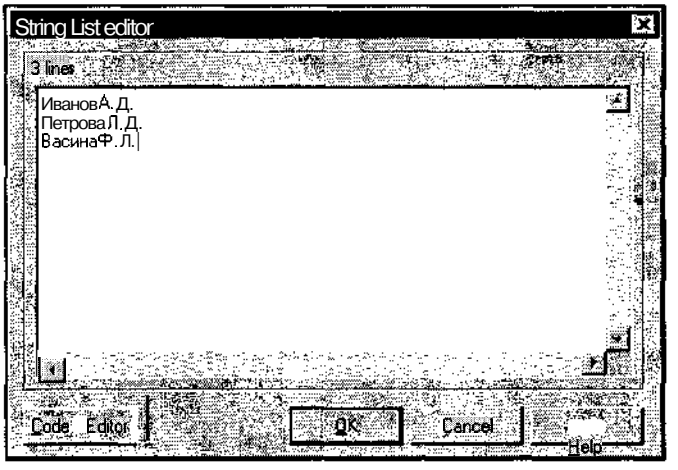


Рис. 4.1. Строковый редактор String List editor

4.2. Использование надписей

Надпись представляет собой нередактируемый текст и часто используется в качестве заголовков для других управляющих элементов, которые не имеют своего свойства `Caption`. Для отображения надписей, в первую очередь, используется компонент Label, называемый также *меткой.* Он представляет собой *простой текст,* который не может быть отредактирован пользователем 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 следующих значений:

- `taLeftJustify` — выравнивание по левому краю;
- `tacenter` — центрирование;
- `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 удобно установить значение True этого свойства, чтобы обрезать лишние слова по ширине компонента Label и перенести 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 *закрашенной* цветом, это определяется свойством Transparent типа Boolean. *Цвет закрашивания* устанавливается свойством `Color`. По умолчанию свойство *Transparent* имеет значение False и надпись не прозрачна. Сделать компонент Label прозрачным может понадобиться в случаях, когда надпись размещается поверх рисунка и не должна закрывать изображение, например, на географической карте.

Так как надпись служит для отображения не редактируемого текста, то иногда Label называют *статическим текстом.* В принципе такое название соответствует назначению этого компонента, однако необходимо учитывать, что есть еще один компонент с таким же названием — staticText. По функциональному назначению Label и staticText практически не отличаются. Однако компонент staticText является оконным компонентом. Другим его отличием от Label является то, что компонент `StaticText` может отображать вокруг себя рамку, вид которой определяется свойством `BorderStyle` типа `TStaticBorderStyle`.

Отобразить не редактируемый текст можно и с помощью других компонентов, например Edit, установив его свойству Readonly значение True.

Пример. Вывод надписи с помощью компонента Edit.

```
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.

4.3. Однострочный редактор

Однострочный редактор, или строка (поле) редактирования представляет собой поле ввода информации, в котором возможно отображение и изменение текста. В Delphi имеется несколько однострочных редакторов, из них наиболее часто используемым является 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 другие. Отметим, что у однострочного редактора отсутствует реакция на управляющие клавиши <Enter> и <Esc>.

Для *проверки информации,* вводимой в редакторы, можно использовать обработчики событий нажатия клавиш, например, обработчик события OnKeyPress.

Пример. Ограничение набора символов, вводимых в редактор.

```
procedure TForm1.Edit1KeyPress(Sender :TObject; var Key :Char);
begin
  if (Key < '0') or (Key > '9') then Key := #0;
end;
```

Здесь для редактора Editl разрешен ввод только десятичных цифр.

В поле редактирования может содержаться одна строка, не имеющая символа конца, поэтому при нажатии клавиши <Enter> не выполняются никакие действия и в строку ничего не вводится. При необходимости программист должен сам кодировать действия, связанные с нажатием клавиши <Enter>. Чаще всего нажатие клавиши <Enter> является признаком окончания ввода информации в поле редактора, и необходимо перейти к другому элементу управления, то есть передать ему фокус ввода, например с помощью метода SetFocus ИЛИ установки значения свойства ActiveControl.

Пример. Задание реакции однострочного редактора на нажатие клавиши <Enter>.

```
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 по нажатию клавиши <Enter> автоматически активизируется очередное поле. Из третьего поля фокус управления автоматически не передается. Передача фокуса управления из разных полей показана 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) управляющему элементу более удобно использование *разделяемого (общего) обработчика* события, связанного с нажатием клавиши <Enter>. Эта процедура может быть общей для всех элементов редактирования и может содержать следующий код:

```
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;
```

Чтобы эта 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.

При нажатии клавиши <Enter> выполняется метод `SelectNext`, *передающий фокус ввода* следующему управляющему элементу. Процедура `SelectNext (CurControl: TWinControl; GoForward, CheckTabStop: Boolean)` имеет три параметра, из которых CurControl указывает оконный управляющий элемент относительно которого выполняется передача фокуса. Параметр GoForward определяет направление передачи фокуса. Если его значение равно True, то фокус получает следующий управляющий элемент, в противном случае — предыдущий управляющий элемент. Параметр CheckTabStop определяет, нужно ли учитывать значение свойства Tabstop управляющего элемента, который должен получить фокус. При значении True параметра управляющий элемент получит фокус, если его свойство Tabstop имеет значение True.

Компонент MaskEdit также представляет собой *однострочный редактор,* но по сравнению с компонентом Edit он дополнительно предоставляет возможность ограничения вводимой 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.

4.4. Многострочный редактор

Для работы *с многострочным текстом* Delphi предоставляет компонент Memo. Многострочный редактор имеет практически те же возможности по редактированию текста, что и однострочные элементы редактирования. Главное отличие этих управляющих элементов заключается в том, что многострочный редактор содержит несколько строк.

Для *доступа ко всему содержимому* многострочного редактора используется свойство Text типа `string`, в этом случае все содержимое компонента Memo представляется одной строкой. При этом символ конца строки, вставляемый при нажатии клавиши <Enter>, представляется двумя кодами `#13#10`, и 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 TString служит для выполнения операций со строками и имеет различные свойства и методы, которые рассмотрены выше. Компо-

нент Memo может использовать возможности этого класса через свое свойство Lines.

Пример. Операции 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 TString 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 LoadFromFile (const FileName: String) класса TStrings. Параметр FileName методов определяет текстовый файл для операций чтения и записи.

Пример. Чтение информации из текстового файла в компонент Memo1.

```
Memo1.Lines.LoadFromFile('C:\TEXT\example1.txt');
```

Пример. Запись информации из компонента мето2 в текстовый файл.

```
Memo2.Lines.SaveToFile('C:\TEXT\example2.txt');
```

Для удобного *просмотра информации* возможно задание в поле редактирования ПОЛОС прокрутки С ПОМОЩЬЮ свойства ScrollBars типа TScrollStyle, принимающего следующие значения:

- ssNone — полосы прокрутки отсутствуют (по умолчанию);
- ssHorizontal — снизу имеется горизонтальная полоса прокрутки;
- ssVertical — справа имеется вертикальная полоса прокрутки;
- 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 уста-

новлено значение True (по умолчанию). В противном случае редактор не реагирует на нажатие клавиши <Enter>.

Аналогичное назначение имеет свойство WantTabs типа Boolean, определяющее реакцию компонента на *нажатие клавиши <Tab>*. Если свойству установлено 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.

Замечание

Даже в случае, когда свойства WantReturns и WantTabs имеют значение False, компонент Memo способен обработать нажатия клавиш <Enter> и <Tab>. Для этого нужно нажать клавишу <Enter> или <Tab> при нажатой клавише <Ctrl>.

Компонент RichEdit представляет собой элемент управления редактирования с форматированием текста и в дополнение к Memo поддерживает такие операции *форматирования текста,* как выравнивание и табуляция текста, применение отступов, изменение гарнитуры и другие. Текст, содержащийся в этом элементе редактирования, совместим с форматом RTF (Rich Text Format), поддерживаемым многими текстовыми процессорами в среде Windows.

4.5. 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 информации, содержащейся в поле ввода, например, для оперативного контроля правильности набора данных. Кроме того, *при модификации данных* редактора свойство Modiefid типа Boolean принимает значение True. Это свойство можно использовать, например, при проверке того, сохранена ли редактируемая информация на диске:

```
if Memo1.Modiefid then begin
   // Здесь располагаются операторы
   // выдачи предупреждения и сохранения информации
   end;
```

Для *указания максимального количества символов,* которые допускается вводить в элемент редактирования, можно использовать свойство MaxLength

типа integer. При этом ограничение на длину текста относится к вводу со стороны пользователя, программно можно ввести большее количество символов, чем это задано в свойстве MaxLength. По умолчанию длина ввода текста пользователем не ограничена (MaxLength = o).

Свойства AutoSelect, SelStart, SelLength И SelText ПОЗВОЛЯЮТ работать С *выделенным (селектированным) фрагментом* текста. Эти свойства доступны не только для чтения, например, в случае анализа текста, вьщеленного пользователем, но и для записи, когда фрагмент выделяется программным способом, например, в процессе поиска или замены информации.

Свойство AutoSelect типа Boolean определяет, будет ли *автоматически выделен текст* в элементе редактирования, когда последний получает фокус управления (по умолчанию имеет значение True) .

Значение свойства SeiText типа string определяет *выделенный фрагмент.* При отсутствии вьщеленного текста значением свойства является пустая строка.

Свойства SelStart и SelLength типа integer задают соответственно начальную *позицию* в строке (отсчет символов в строке начинается с нуля) и *длину* выделенного фрагмента.

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 следует свойству HideSelection типа Boolean установить значение False. Это свойство определяет, будет ли отображаться выделенный текст при потере компонентом фокуса управления. Если свойство HideSelection имеет значение True (по умолчанию), то текст не будет выглядеть выбранным при переходе фокуса на другой элемент управления.

Пример. Операции 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 на строку abcdefgh. В компоненте Edit1 осуществляется поиск строки qwerty. В случае удачного поиска найденный фрагмент выделяется.

Кроме свойств, для *операций с выделенным фрагментом* текста служат такие методы, как SelectAll, CopyToClipBoard и CutToClipBoard.

Метод SelectAll *выделяет весь текст* в элементе редактирования.

Методы CopyToClipBoard и CutToClipBoard соответственно *копируют* и *вырезают* в буфер обмена выделенный фрагмент текста. Например, оператор

```
Memo1.CutToClipBoard;
```

вырезает выделенный фрагмент и помещает его в буфер обмена.

Для работы с *буфером обмена* имеется также метод PasteFromClipBoard, предназначенный для вставки текста из буфера обмена в место текущего расположения курсора в элементе редактирования. Если имеется выделенный фрагмент, то вставляемый текст заменяет его. Более подробно вопросы, связанные с буфером обмена, рассматриваются в главе, посвященной обмену информацией.

Для *проверки данных,* введенных в элемент редактирования, можно использовать событие OnExit, возникающее при окончании ввода, то есть при потере этим элементом фокуса управления.

Пример. Проверка данных, введенных в редактор.

```
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;
```

В редактор Edit1 для нового товара вводится артикул, который не должен быть равен 123 или 456 (в реальных приложениях проверку реализовать сложнее, так как 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, выдается предупреждение, а фокус снова получает редактор Edit1, весь текст которого выделяется.

Основным назначением элементов редактирования является ввод и изменение текста, но 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 вывода');
Memo1.Lines.Add( 'справочной информации');
```

компонент Memo при выполнении программы может получать фокус. При этом в поле ввода отображается курсор, что создает у пользователя иллюзию доступности текста для редактирования.

Подобного не происходит при использовании свойства Enabled. Однако в этом случае поле редактирования становится неактивным, и находящийся в нем текст отображается бледным цветом, что не слишком удобно для чтения. Кроме того, происходит отключение полос прокрутки (при их наличии). Поэтому на практике для отображения нередактируемого текста чаще используется свойство Readonly.

4.6. Использование списков

Список представляет собой упорядоченную совокупность взаимосвязанных элементов, являющихся текстовыми строками. 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 предоставляют возможности по добавлению, удалению и выбору отдельных его элементов (строк).

4.6.1. Простой список

Простой список представляет собой прямоугольную область, в которой располагаются его строковые элементы. Для работы с простым списком Delphi служит компонент ListBox.

Если количество строк больше, чем может их поместиться в видимой области списка, то у области отображения появляется полоса прокрутки. *Ориентация* полосы прокрутки, а также *число колонок,* 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 одним столбцом, и при необходимости автоматически появляется (рис. 4.2, правый список) и исчезает вертикальная полоса прокрутки (рис. 4.2, левый список).

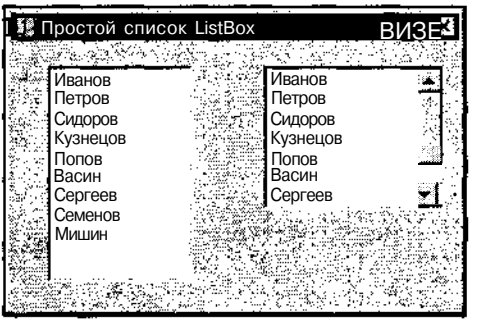


Рис. 4.2. Варианты списков

Если свойство `Columns` имеет значение, большее или равное 1, то в области списка всегда присутствует горизонтальная полоса прокрутки, а элементы разбиваются на такое число колонок (столбцов), чтобы можно было путем прокрутки списка по горизонтали просмотреть все его элементы. При этом в видимой области списка отображается число колонок, определяемое свойством columns. На рис. 4.3 приведен пример двух списков, содержащих 9 фамилий. Для левого списка значение свойства columns равно 2, и в списке одновременно видны 2 колонки, для правого списка значение свойства `Columns` равно 3, и в списке видны все 3 колонки.

Рис. 4.3. Списки С горизонтальной полосой прокрутки

Иногда необходимо, чтобы список одновременно отображал и вертикальную, и горизонтальную полосы прокрутки. В этом случае нужно задать свойству columns нулевое значение, при этом вертикальная полоса прокрутки будет появляться по мере надобности. Для отображения горизонтальной полосы прокрутки следует послать списку сообщение LB_`SetHorizontalExtent`. Третьим параметром сообщения является макси-

мальное 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.

Пример. Список `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 времени выполнения `TopIndex` типа integer.

Стиль простого списка задает свойство style типа `TListBoxStyle`, принимающее следующие значения:

- lbstandard — стандартный стиль (по умолчанию);
- lbOwnerDrawFixed — список с элементами фиксированной высоты, определяемой свойством `ItemHeight`;
- `lbOwnerDrawVariable` - список с элементами, которые могут иметь разную высоту.

Если стиль списка отличен от значения lbstandard, то программист сам отвечает за прорисовку элементов списка. 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).

4.6.2. Комбинированный список

Комбинированный список объединяет поле редактирования и список. Пользователь может выбирать значение из списка или вводить его непосредственно в поле. Для работы с комбинированным списком Delphi служит компонент `ComboBox`.

Список, инкапсулированный в компонент ComboBox, может быть простым и раскрывающимся. Раскрывающийся список в свернутом виде занимает на экране меньше места. На рис. 4.4 показан компонент ComboBox со свернутым и с развернутым списками.

Рис. 4.4. Компонент ComboBox

В отличие от простого, комбинированный список не может иметь горизонтальную полосу прокрутки и из комбинированного списка можно выбрать одно значение.

Свойство style типа TComboBoxStyle определяет *внешний вид* и *поведение* комбинированного списка. Свойство style может принимать следующие значения:

- csDropDown — раскрывающийся список с полем редактирования (по умолчанию). Пользователь может выбирать элементы из списка, при этом выбранный элемент появляется в поле ввода, или вводить (редактировать) информацию непосредственно в поле ввода;
- csSimple — поле редактирования с постоянно раскрытым списком. Чтобы список был виден, необходимо увеличить высоту (свойство Height) компонента ComboBox;
- csDropDownList — раскрывающийся список, допускающий выбор элементов из списка;
- csOwnerDrawFixed — список с элементами фиксированной высоты, которую задает свойство ItemHeight;
- csOwnerDrawVariable — список с элементами, которые могут иметь разную высоту.

Если стиль списка имеет значение csOwnerDrawFixed или csOwnerDrawVariable, то профаммист сам отвечает за прорисовку элементов списка.

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 на раскрывающемся списке 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.

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.

4.6.3. Общие свойства списков

Простой и комбинированный списки во многом похожи друг на друга и имеют много свойств, методов и событий. Основным для списков является рассмотренное ранее свойство items, которое содержит элементы списка и представляет коллекцию (массив) строк.

Элементы списка могут быть отсортированы в алфавитном порядке. Наличие или отсутствие *сортировки* определяет свойство Sorted типа Boolean. При значении False свойства (по умолчанию) элементы в списке располагаются в порядке их поступления в список. Если свойство Sorted имеет значение True, то элементы автоматически сортируются по алфавиту в порядке возрастания.

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 другие, затем следуют буквы латинского алфавита, при этом символы, отличающиеся только регистром, располагаются рядом: сначала — символ в нижнем регистре, затем — верхнем регистре, несмотря на то, что их коды не являются соседними. Последними упорядочиваются буквы русского алфавита, при этом символы, отличающиеся только 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, свойству Sorted нужно присвоить значение 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.

Пример. Отображение номера строки, выбранной в списке ListBox1.

```
Label5.Caption := 'Выбрана ' + IntToStr(ListBox1.ItemIndex) + ' строка';
```

Программист может выбрать элемент списка, установив требуемое значение свойству itemindex. Так, оператор

```
ListBox2.ItemIndex := 3;
```

приводит к выбору 4-й строки списка ListBox2, и это отображается на экране.

По умолчанию в списке можно выбрать один элемент. Для выбора двух и более элементов свойству MultiSelect типа Boolean, *управляющему возможностью выбора нескольких строк,* устанавливается значение True. По умолчанию это свойство имеет значение False.

Замечание

Если свойству MultiSelect установлено значение True, 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 для тех списков, которые не поддерживают 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>.

Так как у комбинированного списка можно одновременно выбирать один элемент, то у него отсутствуют свойства 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';
```

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.

Пример. Программный выбор элементов списка.

```
ListBox1.Selected[1] := true;
ListBoxl.Selected[3] := true;
```

В списке ListBox1 выбираются вторая и четвертая строки.

При выборе элемента списка происходит событие OnClick, которое можно использовать для выполнения *обработки выбранных строк.*

Пример. Обработка события выбора элемента списка.

```
procedure TForm1.ListBox1Click(Sender: TObject) ;
begin
  Label3.Caption := ListBox1.Items[ListBox4.ItemIndex];
end;
```

Надпись Label3 отображает элемент, выбранный в списке ListBoxi.

Глава 5

Кнопки и переключатели

5.1. Работа с кнопками

Кнопки являются управляющими элементами и используются для выдачи команд на выполнение определенных функциональных действий, поэтому часто их называют *командными кнопками*. На поверхности кнопка может содержать текст и/или графический рисунок.

Delphi предлагает несколько компонентов, представляющих различные варианты кнопок:

- стандартная кнопка Button;
- кнопка BitBtn с рисунком;
- кнопка speedButton быстрого доступа.

Как управляющие элементы, эти виды кнопок появились в разное время, но имеют много общего. Различия в облике и функциональных возможностях между разными вариантами кнопок незначительны.

5.1.1. Стандартная кнопка

Стандартная кнопка, или кнопка представлена в Delphi компонентом Button, который является оконным элементом управления.

Кнопка Button на поверхности может содержать надпись, поясняющую назначение и описание действий, выполняемых при ее нажатии.

Основным для кнопки является событие `OnClick`, возникающее *при нажатии* кнопки. 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, то есть кнопка, находящаяся в фокусе, заголовок которой выделен *пунктирным* прямоугольником (рис. 5.1).

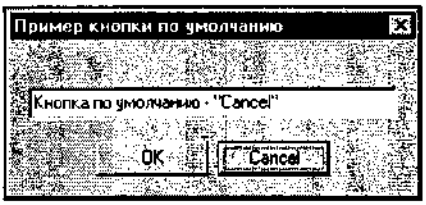


Рис. 5.1. Кнопка по умолчанию — **Cancel**

Если фокус ввода получает не кнопочный элемент управления, например, Edit или Memo, то *кнопкой по умолчанию* становится та, у которой свойству Default типа Boolean установлено значение True. В этом случае кнопка по умолчанию выделяется *черным прямоугольником,* например, кнопка **OK** в диалоговом окне. При размещении в процессе конструирования приложения кнопок на форме (или в другом контейнере, например, Panel) это свойство имеет значение False, то есть выбранных кнопок нет. Если свойству Default программно установить значение True для двух и более кнопок, это не приведет к ошибке, но кнопкой по умолчанию будет являться первая кнопка в порядке обхода при табуляции.

Событие `OnClick` может генерироваться для кнопки и в случае нажатия *клавиши <Esc>,* что обычно делается для кнопок, связанных с отменой какого-либо действия, например, кнопка **Cancel** в диалоговом окне. Чтобы кнопка реагировала на нажатие клавиши <Esc>, необходимо ее свойству cancel типа Boolean установить значение True. При установке свойству `Cancel` значения True для двух и более кнопок кнопкой отмены считается первая в порядке обхода при табуляции кнопка. По умолчанию значение свойства `Cancel` равно False, и никакая кнопка не реагирует на нажатие клавиши <Esc>.

Замечание

Если в фокусе ввода находится не кнопочный управляющий элемент, например, редактор Memo, то он первый получает сообщения о нажатии клавиши и соответственно первым может реагировать на такие клавиши, как <Enter> или <Esc>, обрабатывая их по-своему.

При применении кнопки для *закрытия модального окна* можно использовать ее свойство ModalResult типа TModalResult. Оно определяет, какое значение будет содержать одноименное свойство ModaiResuit формы, когда окно закрывается при нажатии на соответствующую кнопку. Возможными значениями свойства ModaiResuit являются целые числа, некоторые из которых объявлены как поименованные константы:

- mrNone — 0;
- mrOk — idOK (1);
- mrCancel — idCancel (2);
- mrAbort — idAbort (3);
- mrRetry — idRetry (4);
- mrIgnore — idIgnore (5);
- mrYes — idYes (6);
- mrNo — idNo (7) ;
- mrAll — mrNo + 1;
- mrNoToAll — mrAll + 1;
- mrYesToAll — mrNoToAll + 1.

Для поименованных констант в скобках приведены их числовые значения.

Если для кнопки свойству ModaiResuit установлено ненулевое значение, отличное от mrNone (по умолчанию), то при нажатии на кнопку модальная форма автоматически закрывается, и нет необходимости вызова метода Close в обработчике события OnClick этой кнопки.

Пример. Закрытие модальной формы Form2.

```
procedure TForm2.FormCreate(Sender: TObject);
begin
  Button2.ModalResult := mrOK;
  Button3.ModalResult := 123;
end;
procedure TForm2.Button1Click(Sender: TObject);
begin
  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 на любую из кнопок Button1, Button2 или Button3 модальная форма Form2 закрывается. Причем, чтобы закрыть форму не требуются обработчики событий нажатия кнопок Button2 или Button3.

Обычно требуемые значения свойства ModalResult для кнопок устанавливаются при проектировании формы с помощью Инспектора объектов.

5.1.2. 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 дополнительно к стандартному заголовку имеет возможность отображения растрового рисунка (глифа). Видом и размещением изображения на поверхности кнопки BitBtn можно управлять с помощью свойств.

Свойство Glyph типа TBitmap определяет растровое *изображение* кнопки. По умолчанию свойство Glyph имеет значение nil, то есть кнопка не содержит рисунка. Выводимый рисунок может содержать до трех отдельных изображений. Какое именно изображение из трех выводится на кнопке, зависит от текущего состояния кнопки:

- 1-е изображение отображается, если кнопка не нажата (по умолчанию);
- 2-е изображение отображается, если кнопка не активна и не может быть выбрана;
- 3-е изображение отображается, когда кнопка нажата (выполнен щелчок).

При использовании нескольких изображений они должны быть подготовлены и сохранены в файле растрового формата BMP. Подготовить рисунок для кнопки можно, например, с помощью графического редактора Image Editor, входящего в состав Delphi. На рис. 5.2 показано создание рисунка из трех изображений с помощью редактора. Все отдельные изображения в рисунке должны располагаться без промежутков в горизонтальной строке и иметь одинаковую высоту и ширину, как правило, 16 на 16 пикселов. По умолчанию левый нижний пиксел каждого рисунка задает фоновый цвет рисунка. Обычно он устанавливается по цвету поверхности кнопки (значение clBtnFace), и соответственно все пикселы рисунка, имеющие тот же цвет, не будут видны — станут прозрачными. По этой причине фоновый цвет также называют прозрачным. Если цвет фонового пиксела установить, например, желтым, то фон изображения тоже станет желтым.

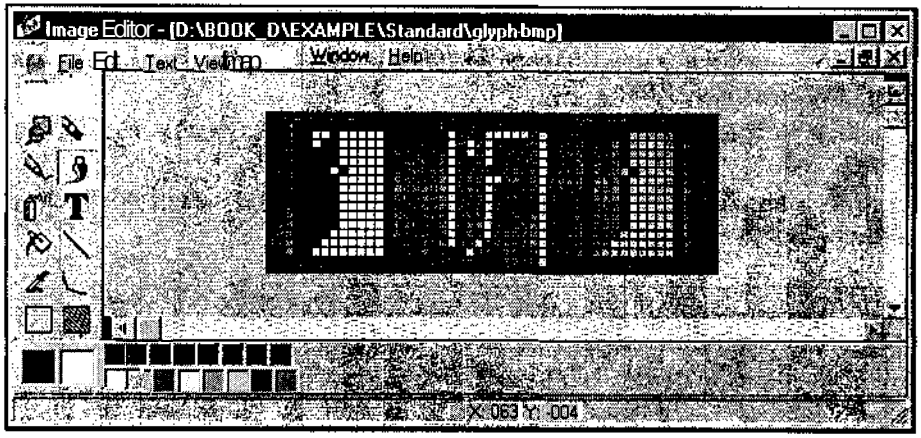


Рис. 5.2. Подготовка рисунка для кнопки

Совместно с Delphi поставляется набор изображений для кнопок, который при установке системы по умолчанию записывается в каталог Program Files\Common Files\Borland Shared\Images\Button.

Delphi предлагает для кнопки BitBtn несколько предопределенных видов (рис. 5.3), выбираемых с помощью свойства Kind типа TBitBtnKind. При выборе какого-либо вида для кнопки на ней отображается соответствующий глиф. При определении вида кнопки могут использоваться следующие константы:

- bkCustom — кнопка имеет выбранное изображение, первоначально изображение отсутствует и его нужно загружать дополнительно;
- bkOK — кнопка имеет зеленую галочку и надпись **OK**; свойству Default кнопки установлено значение True, а свойству ModalResult — значение mrOK;
- bkCancel — кнопка имеет красный знак x и надпись **Cancel;** свойству Cancel кнопки установлено значение True, а свойству ModalResuit — значение mrCancel;
- bkYes — кнопка имеет зеленую галочку и надпись **Yes;** свойству Default кнопки установлено значение True, а свойству ModalResuit — значение mrYes;
- bkNo — кнопка имеет изображение в виде красной перечеркнутой окружности и надпись No; свойству Cancel кнопки установлено значение True, а свойству ModalResuit — значение mrNo;
- bkHelp — кнопка имеет изображение в виде сине-зеленого вопросительного знака и надпись **Help;**
- bkClose — кнопка имеет изображение в виде двери с обозначением выхода и надпись **Close;** при нажатии кнопки форма автоматически закрывается;

- bkAbort — кнопка имеет красный знак x и надпись **Abort;**
- bkRetry — кнопка имеет изображение в виде зеленой стрелки повтора операции и надпись **Retry;**
- bkIgnore — кнопка отображает изображение игнорирования и надпись **Ignore;**
- bkAll — кнопка имеет изображение в виде двойной зеленой галочки и надпись **Yes to All**.

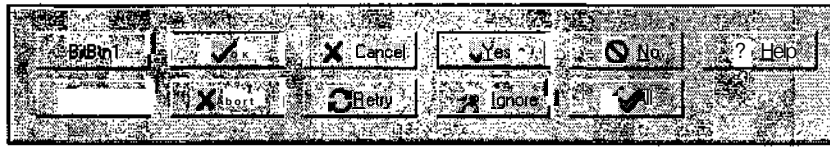


Рис. 5.3. Предопределенные виды кнопок BitBtn

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),** так как в этом случае кнопка не будет выполнять закрепленные за ней действия (в данном случае закрытие окна).

Расположением изображения на поверхности кнопки относительно надписи (рис. 5.4) управляет свойство Layout типа TButtonLayout, принимающее следующие значения:

- blGlyphLeft — изображение слева от надписи (по умолчанию);
- blGlyphRight — изображение справа от надписи;
- blGlyphTop — изображение над надписью;
- blGlyphBottom — изображение под надписью.

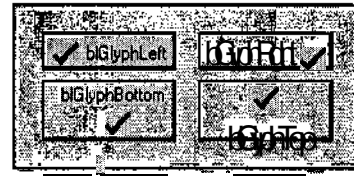


Рис. 5.4. Варианты размещения изображения

5.2. Работа с переключателями

Переключатели (флажки) позволяют выбрать какое-либо значение из определенного множества. Они могут находиться во включенном (установленном) или выключенном (сброшенном) состояниях. Анализ состояния переключателя позволяет программисту выполнять соответствующие операции.

Для работы с переключателями Delphi предоставляет 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, RadioButton — с зависимой фиксацией.

5.2.1. Переключатель с независимой фиксацией

Переключатель *с независимой фиксацией* представлен компонентом CheckBox. Этот переключатель действует независимо от других подобных переключателей.

Для *определения состояния* переключателя используется свойство Checked типа Boolean. По умолчанию оно имеет значение False, и переключатель выключен.

Пользователь может переключать флажок щелчком мыши. Если флажок выключен, то после щелчка он становится включенным, и наоборот. При этом соответственно изменяется значение свойства Checked. Флажок можно переключить и с помощью клавиши <Пробел>, когда компонент checkBox находится в фокусе ввода, и вокруг его заголовка имеется черный пунктирный прямоугольник.

Пример. Анализ состояния независимого переключателя.

```
If CheckBox1.Checked then
   MessageDlg('Время истекло!', mtError, [rabOK], 0);
```

Сообщение время истекло! выдается при включенном состоянии флажка CheckBoxl, управляющего параметром выдачи сообщения об истечении лимита времени.

Флажком можно управлять программно, устанавливая свойству Checked требуемые значения.

Пример. Программное управление независимым переключателем (флажком).

```
CheckBox2.Checked := true;
CheckBox3.Checked := false;
```

Флажок checkBox2 устанавливается в выбранное состояние, а флажок checkBox3 — в невыбранное состояние.

Сделать флажок недоступным для изменения (заблокировать) можно путем установки свойству Enabled значения False:

```
CheckBox1.Enabled := false;
```

После перехода переключателя в заблокированный режим он сохраняет свое состояние, которое было до выполнения блокировки. То есть неактивный переключатель может находиться во включенном и в выключенном состояниях.

Кроме двух состояний (включен — выключен) переключатель может иметь еще и третье состояние — запрещенное, или отмененное. Наличием или

отсутствием *отмененного состояния* управляет свойство AllowGrayed типа Boolean. Если это свойство имеет значение True, то при щелчке мышью переключатель циклически переключается между тремя состояниями: включенный, выключенный и отмененный. В отмененном состоянии переключатель выделяется серым цветом, а в прямоугольнике находится знак галочки.

Замечание

Отображение галочки переключателем в отмененном состоянии способно ввести в заблуждение, так как подобное состояние можно интерпретировать как включенное.

Свойство Checked **имеет значение** True **только для выбранного режима переключателя.**

Для *анализа* и *установки* одного из трех состояний флажка (рис. 5.5) служит свойство state типа TCheckBoxstate. Оно может принимать следующие значения:

- cbchecked — переключатель включен;
- cbUnchecked — переключатель не включен;
- cbGrayed — переключатель запрещен.

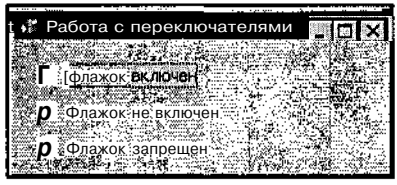


Рис. 5.5. Состояния переключателя CheckBox

При изменении состояния переключателя возникает событие OnClick, независимо от того, в какое состояние переходит переключатель. В обработчике события onclick обычно располагаются операторы, выполняющие проверку состояния переключателя и осуществляющие требуемые действия.

Пример. Процедура обработки события выбора независимого переключателя.

```
procedure TForm1.CheckBox3Click(Sender: TObject);
begin
case CheckBox3.State of
     cbUnchecked: CheckBox3.Caption := 'Переключатель включен';
     cbchecked:   CheckBox3.Caption := 'Переключатель не включен';
     cbGrayed:    CheckBox3.Caption := 'Переключатель запрещен';
     end;
end;
```

Переключатель CheckBox3 при его переключении отображает в заголовке свое состояние.

5.2.2. Переключатель с зависимой фиксацией

Переключатели с зависимой фиксацией представлены компонентом RadioButton, их также называют *кнопками выбора*. Соответствующие элементы управления отображаются в виде круга с текстовой надписью.

Кнопки выбора обычно располагаются по отдельным группам, визуально выделенным на форме. Выбор переключателя является взаимно исключающим, то есть при выборе одного переключателя другие становятся невыбранными. Delphi поддерживает автоматическое группирование переключателей. Каждый переключатель, помещенный в контейнер, включается в находящуюся на нем группу (рис. 5.6). Контейнерами обычно служат такие Компоненты, как форма Form, панель Panel И группа GroupBox.

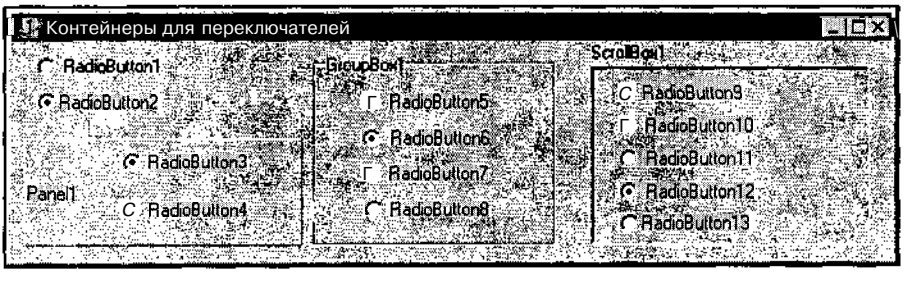


Рис. 5.6. Виды контейнеров для переключателей

При работе с группой один из зависимых переключателей рекомендуется делать выбранным, что можно выполнить при проектировании формы или в процессе выполнения приложения. Например, для приведенной на рис. 6.6 формы это можно выполнить следующим образом:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Все переключатели расположены в разных группах
  RadioButton2.Checked := true;
  RadioButton3.Checked := true;
  RadioButton6.Checked := true;
  RadioButton12.Checked := true;
end;
```

Когда в ipynne выбран один из зависимых переключателей, то, в отличие от независимого переключателя, его состояние нельзя изменить повторным

щелчком. Для отмены выбора зависимого переключателя нужно выбрать другой переключатель из этой же группы.

Замечание

Для компонента RadioButton событие Onclick возникает только при выборе переключателя. Повторный щелчок на переключателе не приводит к возникновению события `OnClick`.

Кроме уже упомянутых элементов — контейнеров, объединяющих зависимые переключатели в группу, Delphi имеет специализированный компонент RadioGroup (рис. 5.7), 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. 5.7. Группа переключателей RadioGroup

Группа переключателей RadioGroup может содержать также другие элементы управления, например, независимый переключатель 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 со 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 можно использовать для выбора отдельного переключателя или для определения, какой из переключателей является выбранным. По умолчанию свойство itemindex имеет значение -1, и не выбран ни один из переключателей.

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 управляющие элементы, например, однострочный редактор Edit или надпись. Label, размещенные в группе RadioGroup.

Пример. Работа 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 RadioGroupl включается 6 переключателей, расположенных в две колонки. Переключатель с заголовком Item 5 становится выбранным.

5.3. Объединение элементов управления

При разработке приложения часто возникает задача объединения, или группирования различных элементов управления. Группирование может понадобиться, например, при работе с переключателями на форме или при создании панели инструментов.

Объединение элементов выполняется с помощью специальных компонентов — контейнеров. *Контейнер* представляет собой визуальный компонент, который позволяет размещать на своей поверхности другие компоненты, 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 пункте рассмотрен специализированный компонент — контейнер RadioGroup, используемый для организации группы зависимых переключателей. Для различных объектов система Delphi предлагает также набор *универсальных контейнеров,* включающий такие компоненты, как:

- группа GroupBox;
- панель Panel;
- область с прокруткой ScrollBox;
- фрейм (рамка) Frame.

Отметим, что форма также является контейнером, с которого обычно и начинается конструирование интерфейсной части приложения. Форма является владельцем всех расположенных на ней компонентов.

5.3.1. 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 верхнем левом углу и объединяющий содержащиеся в нем элементы управления. Например, на рис. 5.6 группа с заголовком **GroupBoxl** используется для объединения зависимых переключателей `RadioButton`.

5.3.2. 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 фаска 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 фаски* соответственно. Каждое из свойств может принимать следующие значения:

- bvNone — фаска отсутствует;
- bvLowered — фаска утоплена;
- bvRaised — фаска приподнята;
- bvspace — действие не известно.

По умолчанию свойство Beveiinner имеет значение bvNone, а свойство `BeveiOuter` – значение bvRaised.

Между фасками может быть промежуток, *ширина* которого в пикселах определяется свойством `BorderWidth` типа `TBorderWidth`. По умолчанию ширина промежутка равна нулю — промежуток отсутствует.

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.

5.3.3. Область прокрутки

Область прокрутки представляет собой окно с возможностью прокрутки информации. Внутри нее размещаются другие управляющие элементы. В Delphi область прокрутки представлена компонентом ScrollBox.

Компонент ScrollBox является элементом управления, область поверхности которого может быть больше той части, которую видит пользователь. Если какой-либо элемент, содержащийся в компоненте ScrollBox, виден не полностью, то автоматически могут появляться полосы прокрутки: горизонтальная (рис. 5.8, *слева),* вертикальная или обе одновременно. При увеличении размеров области полосы прокрутки могут автоматически исчезать (рис. 5.8, *справа),* если они становятся не нужны.

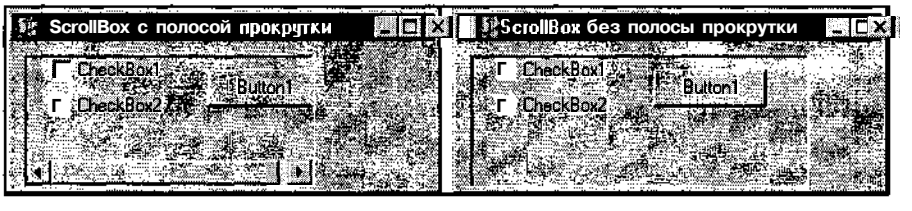


Рис. 5.8. Компоненты ScrollBox

Компонент ScrollBox удобно использовать, например, в случае, когда форма содержит панель инструментов и строку состояния. Если не все управляющие элементы полностью видны в отображаемой области окна, то на нем могут присутствовать полосы прокрутки. Поскольку панель инструментов и строка состояния находятся в клиентской (скроллируемой) облас-

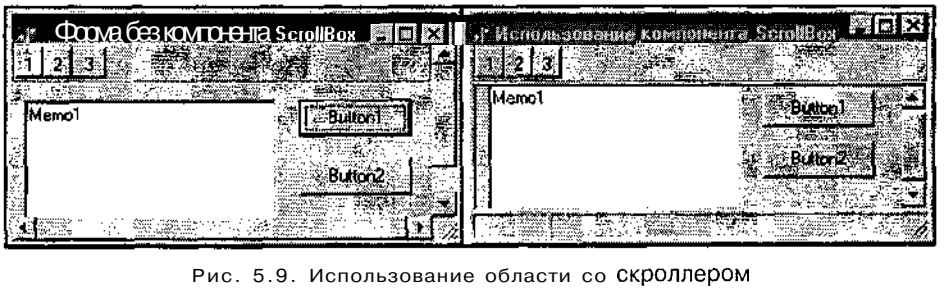


Рис. 5.9. Использование области со скроллером

ти формы, то при появлении вертикальной полосы прокрутки для формы может быть не видна панель инструментов или строка состояния (рис. 5.9, *слева).*

Этого не произойдет, если при проектировании расположить на форме компонент `ScrollBox` и установить его свойству Align значение `alClient`. Область прокрутки займет все место формы, не занятое панелью инструментов и строкой состояния, после чего в ней размещаются другие элементы управления. Теперь, в случае появления полос прокрутки, они будут принадлежать компоненту ScroliBox, обеспечивая доступ ко всем управляющим элементам. В то же время расположенные выше и ниже области с прокруткой панель инструментов и строка состояния будут видны и доступны для выполнения операций (рис. 5.9, *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 свойству Autoscroii установлено значение False, то программист должен отображать полосы прокрутки самостоятельно, управляя свойствами HorzScrollBar И VertScrollBar типа TControlScrollBar.

Для *программного управления* областью прокрутки служит метод `ScrollInView`. Процедура `ScrollInView` (AControl: TControl) автоматически изменяет позиции полос прокрутки так, чтобы интерфейсный элемент, заданный параметром AControl, был виден в отображаемой области.

Глава 6

Использование форм

Форма является важнейшим визуальным компонентом. Формы представляют собой видимые окна Windows и являются центральной частью практически любого приложения. Термины "форма" и "окно" являются синонимами и обозначают одно и то же.

Форма представляется компонентом Form класса `TForm`, на основе формы начинается конструирование приложения. На форме размещаются визуальные компоненты, образующие интерфейсную часть приложения, и системные (невизуальные) компоненты. Таким образом, форма представляет собой компонент Delphi, служащий *контейнером* для всех других компонентов. В связи с отмеченным создание нового приложения начинается с того, что Delphi автоматически предлагает пустое окно — форму `Form1`. В принципе возможно создание и безоконного приложения, однако большинство приложений имеет видимое на экране окно, содержащее его интерфейсную часть.

Каждое приложение может иметь несколько форм, одна их которых является *главной* и отображается первой при запуске программы. 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, указав вызов метода CreateForm создания этой формы первым. Например, для задания Form2 в качестве главной формы можно записать следующий код:

```
Application.Initialize;
Application.CreateForm(TForm2, Form2);
Application.CreateForm(TForm1, Form1);
Application.Run;
```

При конструировании приложения более удобно указать главную форму в окне параметров проекта, вызываемом командой **Project | Options** (Проект | Параметры). Главная форма выбирается в раскрывающемся списке **Main Form** на странице **Form,** после этого Delphi автоматически вносит в файл проекта соответствующие изменения.

Форма может быть модальной и немодальной. *Немодальная* форма позволяет переключиться в другую форму приложения без своего закрытия. *Модальная* форма требует обязательного закрытия перед обращением к любой другой форме приложения.

6.1. Характеристики формы

Как и любой другой визуальный компонент, форма имеет свойства, методы и события, наиболее общие из которых рассмотрены в главе, посвященной интерфейсным элементам. Например, форма имеет такие свойства, как `Caption` и `Color`, событие onKeyPress. Кроме общих для всех визуальных компонентов форма имеет и специфические, определяемые ее особым значением, свойства, методы и события. Часть их характеризует форму как *главный объект приложения,* например, свойство `BorderIcons`, другая часть присуща форме как контейнеру других компонентов, например, свойства `AutoScroll` И`ActiveControl`.

При добавлении новой формы в проект Delphi по умолчанию автоматически создает один экземпляр класса (Forml, Form2 и т. д.), внося соответствующие изменения в файл проекта, например

```
Application.CreateForm(TForm1, Forml);
```

Управлять процессом автоматического создания форм можно, непосредственно редактируя файл проекта, что не рекомендуется делать неопытным программистам, или выполняя настройки в окне параметров проекта (список **Auto-create forms** на странице Form). Если форма переведена из этого списка в список доступных форм проекта (список **Available forms),** то оператор ее создания исключается из файла проекта, и программист в ходе выполнения приложения должен динамически создать экземпляр этой формы.

Для *создания экземпляров форм* служит метод Create. Конструктор Create создает экземпляр класса формы, сам класс формы обычно предварительно описывается при конструировании приложения, и соответственно для формы существуют файлы формы (DFM) и программного модуля (PAS).

Пример. Создание экземпляра формы.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // Форма создается, однако на экране не отображается
  Form2 := TForm2.Create(Application);
  Form2.Caption := 'Новая форма';
end;
```

Форма Form2 принадлежит объекту приложения и имеет заголовок Новая форма.

При создании и использовании формы происходят следующие события типа TNotifyEvent, указанные в порядке их возникновения:

- OnCreate;
- OnShow;
- OnResize;
- OnActivate;
- OnPaint.

Событие OnCreate при создании формы возникает один раз, остальные события происходят каждый раз при отображении формы, при ее активизации и прорисовке соответственно.

В обработчик события OnCreate обычно включается код, устанавливающий начальные значения свойств формы, а также ее управляющих элементов, то есть выполняющий начальную инициализацию формы в дополнение к установленным на этапе разработки приложения параметрам. Кроме того, в обработчик включаются дополнительные операции, которые должны происходить однократно при создании формы, например, чтение фамилий сотрудников и загрузка их в список.

Пример. Процедура Обработки СОбыПИЯ OnCreate формы Form2.

```
procedure TForm2.FormCreate(Sender :TObject);
begin
Form2.Caption := 'Пример формы';
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, а кнопка Button3 блокируется.

Из всех созданных форм Delphi при выполнении приложения автоматически устанавливает видимой главную форму, для этого свойству visible этой формы устанавливается значение True. Для остальных форм значение этого свойства по умолчанию равно False, и после запуска приложения другие формы первоначально на экране не отображаются. Программист после создания форм по мере необходимости сам должен обеспечивать их отображение и скрытие в процессе работы приложения, управляя свойством visible.

Замечание

Даже если форма не видима, ее компонентами можно управлять, например, из других форм.

Другим способом *управления видимостью* форм на экране являются соответственно методы Show и Hide. Процедура Show отображает форму в *немодальном* режиме, при этом свойству Visible устанавливается значение True, а сама форма переводится на передний план. Процедура Hide скрывает форму, устанавливая ее свойству visible значение False.

Если окно уже видимо, то вызов метода Show переводит форму на передний план и передает ей фокус управления.

Пример. Отображение и скрытие форм.

```
procedure TForm1.btnShowFormsClick(Sender :TObject);
begin
  Form2.Visible := true;
  Form3.Show;
end;

procedure TForm1.btnHideFormsClick(Sender :TObject);
begin
  Form2.Visible := false;
  Form3.Hide;
end;
```

Нажатие кнопок btnShowForms и btnHideForms, расположенных на форме Form1, приводит соответственно к отображению и скрытию форм (удалению С экрана) Form2 И Form3.

В момент *отображения* формы на экране, когда ее свойство visible принимает значение True, возникает событие OnShow. Соответственно при *скрытии* формы, когда ее свойство visible принимает значение False, возникает событие OnHide.

Каждый раз при *получении фокуса* ввода формой, например, при нажатии кнопки мыши в области формы, происходит ее активизация и возникает событие onActivate. При *потере фокуса* формой возникает событие OnDeActivate.

Событие onPaint возникает при необходимости *перерисовки* формы, например, при активизации формы, если до этого часть ее была закрыта другими окнами.

Для закрытия формы используется метод Close, который удаляет форму с экрана. Процедура close делает форму невидимой, не уничтожает создан-

ный ее экземпляр, и форма может быть снова вызвана на экран, например, С ПОМОЩЬЮ МеТОДОВ Show ИЛИ ShowModal.

В случае закрытия главной формы прекращается работа всего приложения.

Пример. Закрытие формы.

```
procedure TForm2.btnCloseClick(Sender: TObject);
begin
  Form2.Close;
end;
```

Кнопка btnClose закрывает форму Form2. Для этой кнопки полезно задать соответствующий заголовок (свойство Caption), например, **Закрыть.**

Уничтожение формы можно выполнить с помощью метода Free, после чего работа с этой формой невозможна, и любая попытка обратиться к ней или ее компонентам вызовет исключительную ситуацию (ошибку). Необходимость уничтожения формы может возникнуть при оформлении заставок или при разработке больших приложений, требующих экономии оперативной памяти.

Пример. Удаление экземпляра формы.

```
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 происходят 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 можо закрыть. Если установить параметру CanClose значение False, то форма остается от-

крытой. Такую возможность можно использовать, например, для подтверждения закрытия окна или проверки, сохранена ли редактируемая информация на диске. Событие OnCloseQuery вызывается всегда, независимо от способа закрытия формы.

Пример. Процедура 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 "Вы хотите закрыть форму?" и двумя кнопками — **Yes** и No. В случае подтверждения нажатием кнопки **Yes** форма закрывается, в противном случае закрытия 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 минимизируется.

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.

Замечание

Когда закрывается главная форма приложения, все остальные окна закрываются без вызова события onClose.

Пример. Процедура закрытия формы.

```
procedure TForm2.FormClose(Sender: TObject; var Action: TCioseAction);
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.

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 управляющих элементов окна, не имеющих свойства Align.

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 (заставки).

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

Видимое отличие между диалоговой и обычной формой 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 другую сторону. Программно при выполнении приложения для формы любого стиля можно устанавливать различные допустимые размеры окна, а также изменять их.

Пример. Изменение размеров формы.

```
procedure TForm2.btnResizeFormClick(Sender: TObject);
```

```
begin
  Form2.Width := Form2.Width + 100;
end;
```

При нажатии на кнопку btnResizeForm ширина формы Form2 увеличивается на 100 пикселов, даже если для этой формы 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 и 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. Возможный *набор кнопок* определяет свойство Bordericons типа TBorderIcons, которое может принимать комбинации следующих значений:

- bisystemMenu — окно содержит кнопки системного меню;
- biMinimize — окно содержит кнопку минимизации (свертывания);
- biMaximize — окно содержит кнопку максимизации (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 пиктограмма приложения, при щелчке на которой и появляются команды этого меню, а в области заголовка справа имеется кнопка закрытия формы (рис. 6.1).

Рис. 6.1. 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 максимизации и минимизации, то не отображается кнопка справки. Возможность появления кнопок также зависит от стиля формы. Например, отображение кнопок максимизации и минимизации

зации возможно только для обычной формы и формы панели инструментов с возможностью изменения размеров.

Замечание

Обычно стиль формы и набор кнопок заголовка задаются на этапе разработки приложения через Инспектор объектов. При этом на проектируемой форме всегда видны обычная рамка и три кнопки (минимизации, максимизации и закрытия формы), независимо от значения свойств `FormStyle` и `BorderIcons`. Заданные стиль формы и набор кнопок становятся видимыми при выполнении программы.

Форма включает клиентскую и неклиентскую области. *Неклиентская* область занята рамкой, заголовком и строкой главного меню. В *клиентской* области обычно размещаются различные управляющие элементы, происходит вывод текста или отображение графики. При необходимости в клиентской области могут появляться полосы прокрутки. Аналогично тому, как свойства width и Height определяют размеры всей формы, свойства ClientWidth И ClientHeight типа Integer задают соответственно ширину И высоту в пикселах клиентской части формы.

Пример. Вывод в заголовке формы информации о размерах ее клиентской области.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
Form1.Caption := 'Клиентская область — ' +
   IntToStr(Form1.ClientWidth) +' x '+ IntToStr(Form1.ClientHeight);
end;
```

Отображаемое формой *меню* задается свойством Menu типа TMainMenu. При разработке приложения размещение компонента MainMenu главного меню на форме вызывает автоматическое присвоение нужного значения (`Menu:=MainMenu1`) свойству Menu формы, что отображается в Инспекторе объектов.

Каждая форма отображает в левой стороне области заголовка свою *пиктограмму,* определяемую свойством `Icon` типа `TIcon`. Если форма не является главной в приложении, то эта пиктограмма отображается при минимизации формы. Для любой формы свойство `Icon` можно задать с помощью Инспектора объектов или динамически (при выполнении приложения). Если пиктограмма не задана, то форма использует пиктограмму, указанную в свойстве `Icon` объекта Application. Последняя выводится также при минимизации и отображении в панели задач Windows значка главной формы приложения.

Пример. Динамическая загрузка пиктограммы.

```
procedure TForm1.FormCreate(Sender: TObj ect);
```

```
begin
    Form1.Icon.LoadFromFile('c:\Picture1.ico');
end;
```

Здесь при создании формы Forml пиктограмма загружается из файла c:\Picturel.ico.

Размещение и *размер* формы при отображении определяет свойство Position типа TPosition. Свойство имеет следующие основные значения:

- 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 тот, 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 программист не имеет возможность контролировать эти параметры. В связи с отмеченным это значение 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.

Свойство Active типа Boolean позволяет определить *активность* формы. В любой момент времени активной может быть одна форма, при этом ее заголовок выделяется цветом, чаще всего синим. Если свойство Active имеет значение True, то форма активна (находится в фокусе ввода), если False — то неактивна. Это свойство относится ко времени выполнения и доступно для чтения. Если требуется активизировать форму программно, следует использовать свойство windowstate или метод show (ShowModal).

Свойство windowstate типа Twindowstate определяет состояние отображения формы и может принимать одно из трех значений:

- wsNormal — обычное состояние (по умолчанию);
- wsMinimized — минимизация;
- wsMaximized — максимизация.

Пример. Управление 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 на форме Form1 соответственно МИнимизируют и восстанавливают нормальное состояние формы Form2.
Форма, для которой изменяется состояние отображения на экране, предварительно должна быть создана методами CreateForm или Create. Если форма не создана, то при работе с ней произойдет исключительная ситуация, несмотря на то, что переменная формы объявлена в модуле. Если форма создана, но не отображается на экране, то изменения ее состояния (свойство WindowState) вступают в силу, однако пользователь этого не видит до тех пор, пока форма не будет отображена на экране.

Форма, как контейнер, содержит в себе другие управляющие элементы. Оконные элементы управления (потомки класса TWinControl) могут получать *фокус ввода* (фокус). Свойство ActiveControl типа TWinContro i определяет, какой элемент на форме находится в фокусе управления. Для выбора элемента, находящегося в фокусе ввода (активного элемента), можно устанавливать этому свойству нужное значение в процессе выполнения программы, например:

```
Form1.ActiveControl := Edit2;
```

Эту же операцию выполняет метод setFocus, который устанавливает фокус ввода на оконный элемент управления, например:

```
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 AutoScroil имеет значение True (по умолчанию), то полосы прокрутки появляются и исчезают автоматически, и программист может не заботиться об этом. Необходимость в полосах прокрутки может возникнуть, если пользователь уменьшит размеры формы так, что не все элементы управления будут полностью видимы. В случае, когда программист установит свойству AutoScroil значение False, он сам должен решать проблему просмотра информации, управляя горизонтальной и вертикальной полосами Прокрутки через СВОЙСТВа HorzScrollBar И VertScrollBar типа TControlScrollBar формы.

Для программного *управления полосами прокрутки* можно использовать метод ScrollInView. Процедура ScrollInView (AControl: TControl) автома-

тически изменяет позиции полос прокрутки так, чтобы заданный параметром AControi управляющий элемент 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 (находящемуся в фокусе ввода). При установке свойству KeyPreview значения True форма получает сообщения о нажатии клавиш в первую очередь и может на них реагировать, что обычно используется для обработки комбинаций клавиш, независимо от активности управляющих элементов формы.

Замечание

Форма не может обрабатывать нажатие клавиши <Tab> в связи с ее особым назначением.

Пример. Обработка нажатия клавиш.

```
//Не забудьте установить свойству KeyPreview 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.

Пример. Приложение — цифровые часы.

Приложение похоже на часы, входящие в состав Windows, и состоит из одной формы (рис. 6.2), на которой размещены надпись `Label1`, таймер `Timer1` И главное меню `MainMenu1`.

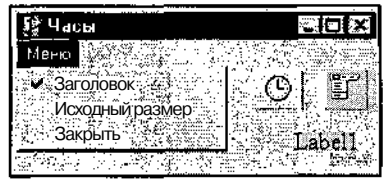


Рис. 6.2. Вид формы при разработке

Таймер используется для отсчета времени, его интервал задан равным юоо, и событие `OnTimer` генерируется один раз в секунду. В обработчике этого события текущее значение времени отображается в надписи `Label1`.

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, а форма отображает заголовок и меню (рис. 6.3, слева). Если выбрать пункт **Заголовок,** то галочка пропадет, а заголовок и меню становятся невидимыми, изменяется также граница окна (рис. 6.3, справа). В этом случае пользователь не сможет изменять размеры окна и перемещать его по экрану. Для восстановления прежнего вида формы нужно дважды щелкнуть мышью на ней или на надписи Label1.

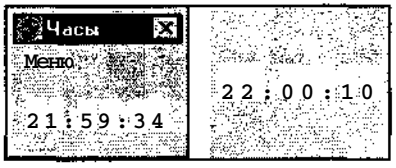


Рис. 6.3. Варианты формы

Ниже приводится код модуля uClock формы Form1. Для наглядности многие свойства компонентов получают свои значения при создании формы в обработчике события OnCreate. В действительности эти действия проще выполнить при разработке приложения с помощью Инспектора объектов.

```
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;
    mnuinitiaisize: TMenuItem;
    mnuClose: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
```

```
    procedure FormResize(Sender: TObject);
    procedure mnCloseClick(Sender: TObject);
    procedure ranSizeOClick(Sender: TObject);
    procedure mnCaptionClick(Sender: TObject);
    procedure FormDblClick(Sender: TObject);
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 через инспектор 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 меню "Заголовок"
```

```
// 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 изменять 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 установлено ограничение на минимальную высоту и ширину формы. Следует учитывать, что это свойство ограничивает минимальные размеры формы, а не ее клиентской области. При выборе пункта меню **Исходный размер** форма восстанавливает размеры клиентской области, заданные при разработке.

Пиктограмма 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 свойству FormStyle задано значение fsStayOnTop.

6.2. Организация взаимодействия форм

Если одна форма выполняет какие-либо действия с другой формой, то в списке uses раздела implementation (или interface) модуля первой формы должна быть ссылка на модуль второй формы.

Пример. Организация взаимодействия форм.

Приложение включает в свой состав две формы — Form1 и Form2, для которых имеются модули Unit1 и Unit2 соответственно. Ниже 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);
  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.Show;
end;

end.
```

При нажатии на кнопку Button1 на первой форме на экране отображается вторая форма, до этого невидимая. Так как из модуля первой формы осуществляется операция со второй формой, то в разделе implementation первого модуля помещен код uses Unit2.

Ссылку на модуль другой формы можно устанавливать программно, но Delphi позволяет выполнить автоматизированную вставку ссылки следующим образом. При задании команды **File | Use Unit** (Файл | Использовать модуль) появляется диалоговое окно **Use Unit** (Выбора модуля) (рис. 6.4). После выбора нужного модуля и нажатия кнопки **OK** ссылка на него добавляется автоматически.

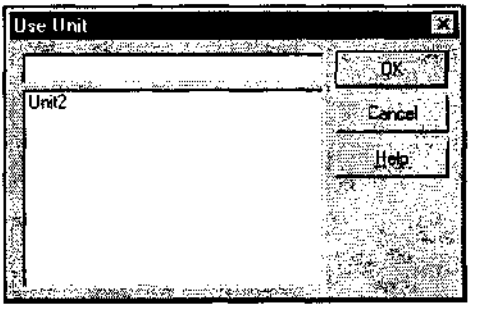


Рис. 6.4. Окно выбора модуля

Если ссылка на требуемый модуль отсутствует, то при компиляции программы появляется диалоговое окно **Information** (Информация) (рис. 6.5). В нем сообщается о том, 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.**

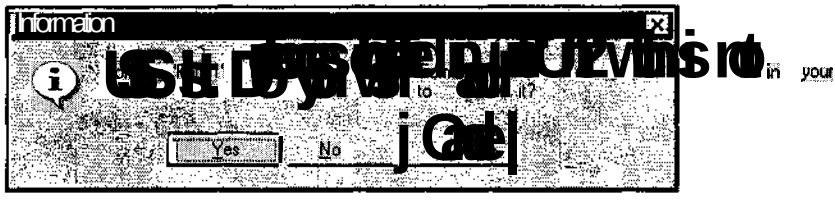


Рис. 6.5. Диалог добавления ссылки на модуль

Форма может выполнять различные операции не только с другой формой, но и с отдельными ее компонентами. В этом случае также нужна ссылка на модуль другой формы.

Пример. Обращение к компоненту другой формы.

```
uses Unit2;
...
procedure TForm1.Button2Click(Sender: TObject);
begin
  Label1.Caption := Form2.Edit1.Text;
end;
```

При нажатии кнопки Button2 формы Formi в надписи `Label1` отображается текст редактора `Edit1`, расположенного на форме Form2.

Замечание

Можно выполнять операции с компонентами формы, минимизированной или невидимой на экране. Однако в любом случае форма уже должна быть создана перед выполнением любых операций с ней.

6.3. 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, то Windows блокирует эту попытку и выдает предупреждающий сигнал. Запрет перехода в другую форму при незакрытой модальной форме относится

только к текущему приложению, и пользователь может активизировать любое другое приложение Windows.

Отметим, что программно возможен доступ к компонентам любой созданной формы приложения, несмотря на наличие в данный момент времени открытой модальной формы.

Модальные формы часто называют *диалогами,* или *диалоговыми панелями,* существуют и немодальные диалоги. 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 диалогами,* для работы с ними Delphi предлагает специальные компоненты. Они рассматриваются в этой главе отдельным пунктом. Типичным примером модальной диалоговой формы является диалоговое окно **About** (О программе).

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`.

Пример. Отображение модальной формы.

```
procedure TForm1.mnuAboutClick(Sender: TObject);
begin
  fmAbout.ShowModal;
end;
```

Выбор пункта меню `mnuAbout` приводит к отображению формы fmAbout в модальном режиме. Пункт меню может иметь заголовок (например, **О программе),** устанавливаемый с помощью свойства `Caption`. Пользователь может продолжить работу с приложением, только закрыв эту модальную форму.

Многие формы можно отображать и в немодальном режиме, например, следующим образом:

```
fmAbout.Show;
```

При закрытии модальной формы функция `ShowModal` возвращает значение свойства `ModalResult` типа `TModalResult`, присваиваемое свойству при этом. Возможные значения этого свойства рассматриваются при описании кнопок. Напомним, что метод show является процедурой и результат не возвращает.

При закрытии любая форма возвращает код результата. Этот код обычно представляет интерес при организации диалогов. После закрытия диалога возвращаемый код результата можно проанализировать и выполнить соответствующие действия. В общем случае в зависимости от кода результата

закрытия модальной формы программируется разветвление на несколько направлений.

Пример. Управление диалоговой формой.

```
// Процедура находится в модуле формы Form1
procedure TForm1.btnDialogClick(Sender: TObject);
var rez :TModalResult;
begin
// Вызов модальной формы (диалога)
rez := fmDialog.ShowModal;
// Анализ способа закрытия модальной формы (диалога)
if rez = mrOK then
   MessageDlg('Диалог принят.', mtInformation, [mbYes], 0);
if rez = mrCancel then
   MessageDlg('Диалог отменен.', mtlnformation, [mbYes], 0);
end;

// Процедуры находятся в модуле формы fmDialog
// Закрытие формы и установка значения mrOK коду результата
procedure TfmDialog.btnCloseClick(Sender: TObject);
begin
ModaiResuit := mrOK;
end;

// Закрытие формы и установка значения mrCancel коду результата
procedure TfmDialog. btnCancelClick(Sender: TObject);
begin
ModaiResuit := mrCancel;
end;
```

Кнопка btnDialog формы Forml открывает диалоговую форму fmDialog. После закрытия диалога кнопкой `btnClose` или `btnCancel` выполняется анализ кода результата закрытия и вывод на экран.

Как правило, управление кодом результата диалога выполняется не программно (через свойство ModaiResuit формы), а с помощью кнопок. Чаще всего диалоговая форма содержит кнопки подтверждения и отмены выполненных операций. Кнопка подтверждения диалога в зависимости от назначения может иметь разные названия, такие как **OK**, **Ввод, Открыть, Yes.** Кнопка отмены диалога часто имеет названия **Отмена** и **Cancel.**

Как отмечалось ранее, закрыть форму можно, используя свойство ModalResult кнопки. Если свойство имеет значение, отличное от mrNone, то при нажатии на кнопку форма автоматически закрывается. При закрытии форма в качестве результата возвращает значение, определяемое свойством ModaiResuit кнопки, закрывшей эту форму.

Пример. Задание кнопок закрытия формы.

```
procedure TfmDialog.FormCreate(Sender: TObject);
begin
fmDialog.BorderStyle := bsDialog;
btnOK.Caption := 'OK';
btnOK.Default := true;
btnOK.ModalResult := mrOK;
btnCancel.Caption := 'Îòìåíà';
btnCancel.Cancel := true;
btnCancel.ModalResult := mrCancel;
end;
```

В приведенной процедуре 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 через Инспектор 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 ModaiResuit. Скрытие формы методом Hide не изменяет значение свойства ModaiResuit.

В принципе разработчик может самостоятельно создать любую модальную форму, однако полезно учитывать, что для выполнения типовых действий Delphi предлагает ряд предопределенных диалогов. Наиболее простые диалоги реализуются с помощью специальных процедур и функций, в более общих случаях удобно использовать специальные компоненты — стандартные диалоги. Кроме того, ряд диалоговых форм расположены на странице **Dialogs** (Диалоги) в Хранилище объектов.

6.4. Процедуры и функции, реализующие диалоги

Процедура ShowMessage, функции MessageDlg И MessageDlgPos отображают окно (панель) *вывода* сообщений; функции InputBox и InputQuery отображают окно (панель) для *ввода* информации. Рассмотрим ShowMessage и MessageDlg.

Процедура ShowMessage (const Msg: String) отображает *ОКНО сообщения* С кнопкой OK. Заголовок содержит название исполняемого файла приложения, а строка Msg выводится как текст сообщения.

Пример. Отображение простейшего окна сообщений.

Рассмотрим отображение простейшего окна сообщений (рис. 6.6) из программы dlgwin.exe.

```
procedure TForm1.btnDialog1Click(Sender: TObject);
begin
  ShowMessage('Простейшее диалоговое окно');
end;
```

Рис. 6.6. Простейшее окно сообщения

Функция MessageDlg (const Msg: String; AType: TMsgDlgType; AButtons: TMsgDlgButtons; HelpCtx: Longint): Word **отображает *ОКНО сообщений*** в центре экрана и позволяет получить *ответ* пользователя. Параметр Msg содержит отображаемое сообщение.

Окно сообщений может иметь различный тип 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 с надписью **Yes;**
- mbNo — кнопка с надписью No;
- mbOK — кнопка с надписью OK;
- mbCancel — кнопка с надписью **Cancel;**
- mbHelp — кнопка с надписью **Help;**
- mbAbort — кнопка с надписью **Abort;**
- mbRetry — кнопка с надписью **Retry;**
- mbIgnore — кнопка с надписью **Ignore;**
- mbAll — кнопка с надписью **All.**

Для параметра AButtons имеются две константы mbYesNoCancel и mbOKCancel, определяющие предопределенные наборы кнопок:

- mbYesNoCancel = [mbYes, mbNo, mbCancel];
- mbOKCancel = [mbOK, mbCancel].

При нажатии любой из этих кнопок, кроме кнопки **Help,** закрывается диалог и возвращается функцией MessageDlg модальный результат (свойство ModalResult), который путем анализа можно использовать для управления выполнением приложения.

Параметр HelpCtx определяет контекст (тему) справки, которая появляется во время отображения диалога при нажатии пользователем клавиши <F1>. Обычно значение этого параметра равно нулю.

Пример. Использование функции MessageDig.

```
procedure TForm1.btnTestDateClick(Sender: TObject);
var rez :TModalResu.lt;
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 if edtDate.CanFocus then edtDate.SetFocus;
   end;
end;
```

При нажатии на кнопку `btnTestDate` производится простейшая проверка даты. Код даты вводится в поле редактирования `edtDate`, размещенное на форме. Если длина даты меньше допустимой, выдается предупреждение и запрос на автоматическую коррекцию. При утвердительном ответе пользователя в поле даты записывается текущая дата, при отрицательном — фокус передается полю ввода даты.

6.5. Стандартные диалоги

В Delphi имеется десять компонентов, находящихся на странице **Dialogs** Палитры компонентов (рис. 6.7) и реализующих диалоги общего назначения. Эти диалоги 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 нижеперечисленных диалогов. Поскольку стандартные диалоги определяются средой Windows, то их внешний вид, в том числе язык интерфейсных элементов, зависит от версии установленной на компьютере Windows.

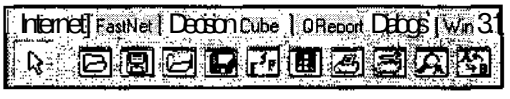


Рис. 6.7. Страница **Dialogs** Палитры компонентов

На странице **Dialogs** Палитры компонентов содержатся следующие компоненты, реализующие стандартные диалоги:

- `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.

Для использования стандартного диалога соответствующий ему компонент должен быть помещен на форму и его свойствам установлены нужные зна-

чения. После этого следует связать вызов диалога с каким-либо событием. Чаще всего таким событием является выбор пункта меню или нажатие кнопки.

Для *вызова* любого стандартного диалога используется метод Execute — функция, возвращающая логическое значение. При закрытии диалога кнопкой **OK (Открыть** или **Сохранить)** функция Execute возвращает значение True, а при отмене диалога — значение False.

После закрытия стандартного диалога он 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`).

Рассмотрим диалоги выбора имени файла, которые используются в процессах открытия и сохранения файла. Часто эти диалоги называют диалогами открытия/сохранения файла. Хотя диалог позволяет только выбрать имя файла, последующее же открытие или сохранение этого файла программируется дополнительно.

Компонент `OpenDialog` реализует диалог *открытия файла,* при запуске этого диалога появляется окно (рис. 6.8) в котором можно выбрать имя открываемого файла. В случае успешного закрытия диалога (нажатием кнопки **Open)** в качестве результата возвращается выбранное имя файла.

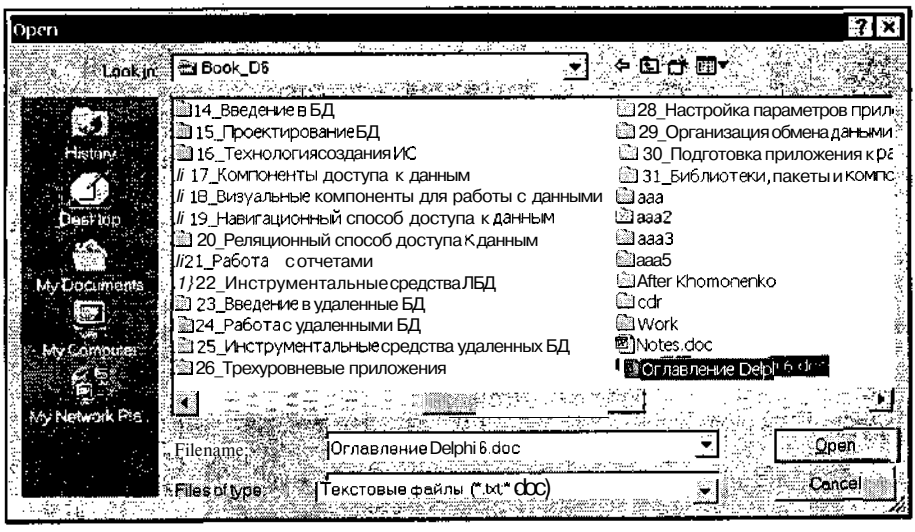


Рис. 6.8. Диалог открытия файла

Компонент SaveDialog предлагает стандартный диалог *сохранения файла*, который отличается от диалога открытия файла только своим заголовком.
Основными свойствами компонентов OpenDialog и SaveDialog являются следующие:

- FileName типа string — указывает имя и полный путь файла, выбранного в диалоге. Имя файла отображается в строке редактирования с названием 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 окна диалога. Если каталог не задан, то отображается содержимое текущего каталога;
- DefaultExt типа string — задает расширение, автоматически подставляемое к имени файла, если пользователем расширение имени не указано;
- Filter типа string — задает маски имен файлов, отображаемых в раскрывающемся списке под названием **Files of type.** В окне диалога видны имена файлов, которые совпадают с указанной маской (на рис. 6.8 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 Fiiterindex имеет значение 1, и используется первая маска;
- Options типа TOpenOptions — используется для настройки параметров, управляющих внешним видом и функциональными возможностями диалога. Каждый параметр (флажок) может быть включен или выключен. Свойство Options имеет около двух десятков параметров, к числу важнейших из них можно отнести следующие:

  - 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 символа для расширения);
  - ofOldStyleDialog — создает окно диалога в стиле Winows 3.11.

Фильтр представляет собой последовательность значений, разделенных знаком I. Каждое значение состоит из описания и маски, также разделенных знаком I. *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 ;.

Пример. Формирование фильтра.

```
OpenDialog1.Filter :='Текстовые файлы|*.txt;*.doc|Все файлы|*.*';
```

Здесь фильтр формируется с двумя масками — маской для текстовых файлов и маской для всех файлов. В примере под текстовыми понимаются файлы типов DOC и TXT.

Так как в раскрывающемся списке диалога отображается только описание фильтра, то для удобства целесообразно включить в описание и маску, например, следующим образом:

```
OpenDialog1.Filte r:= 'Текстовые файлы *.txt;*.doc|*.txt;*.doc'
      + '|Все файлы *.*|*.*';
```

Фильтр обычно формируется при проектировании приложения. Для этого из Инспектора объектов щелчком в области значения свойства Filter вызывается Редактор фильтра **(Filter Editor).** Рабочее поле Редактора фильтра представляет таблицу (рис. 6.9), состоящую из двух колонок с заголовками **Filter** Name (Имя фильтра) и **Filter** (Фильтр). Значения фильтра вводятся построчно, при этом в левой колонке указывается описание фильтра, в правой — соответствующая маска.

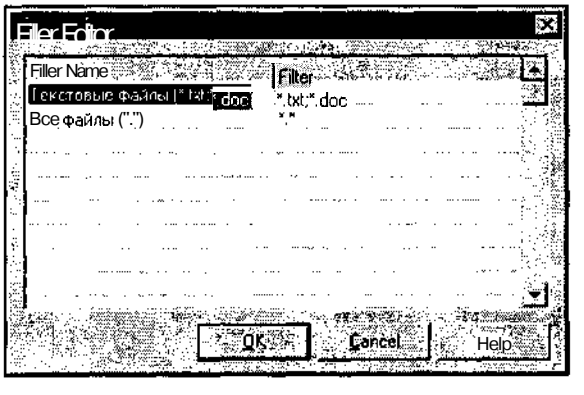


Рис. 6.9. Редактор фильтра

Стандартные диалоги *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 *отвергнутым,* и функция возвращает значение False.

Пример. Использование стандартного диалога OpenDiaiog.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
if OpenDialog1.Execute then
   Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
end;
```

При нажатии на кнопку Button2 появляется диалог OpenDialogl 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 OpenDialogl открытия файла не происходит.

Компоненты OpenPictureDialog И SavePictureDialog вызывают стандартные диалоги *открытия* и *сохранения графических файлов.* Эти стандартные диалоги отличаются от OpenDiaiog и SaveDialog видом окон и установленными значениями свойства Filter.

6.6. 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. Выделим следующие шаблоны форм:

- страница New:
  - пустая форма **Form;**
- ❒ страница **Forms:**
  - справочное окно About Box;
  - форма **Dual List Box** с двумя списками;
  - форма **Quick Report List** отчета;
  - форма **Quick Report Master/Detail** отчета;
- ❒ страница **Dialogs:**
  - диалоговое окно **Dialog with Help** с кнопкой **Help** (два варианта формы, различающиеся расположением кнопок);
  - диалоговое окно **Password Dialog** для ввода пароля;
  - обычный диалог **Standard Dialog** (два варианта формы, различающиеся расположением кнопок).

При добавлении в проект новой формы Delphi автоматически вставляет пустую форму Form, к которой разработчик добавляет необходимые интерфейсные компоненты. Новая форма добавляется при выборе пункта меню **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** (Диалог ввода пароля) (рис. 6.10). Эта форма имеет заголовок **Password Dialog,** имя `PasswordDlg` и содержит надпись `Label1` и две **кнопки закрытия Диалога (ОКНа):** `OKBtn` И `CancelBtn`.

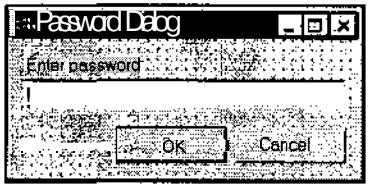


Рис. 6.10. Шаблон диалогового окна **Password Dialog**

Для кнопок закрытия диалога установлены следующие значения свойства `ModalResult`:

- `OKBtn` — `mrOK`;
- `CancelBtn` — `mrCancel`.

Программист может изменить значения компонентов, например, заменив английский текст на русский, или расположить на форме новые компоненты.

После этого окно **Password Dialog** можно вызвать на экран с помощью метода `ShowModal` следующим образом:

```
PasswordDlg.ShowModal;
```

Аналогично можно использовать шаблоны других форм из хранилища объектов. Кроме того, возможно сохранение в хранилище своих форм, которые планируется использовать в других проектах.

Кроме шаблонов форм, в Хранилище объектов находятся мастера (wizards) — специальные программы-утилиты, позволяющие достаточно удобно создавать формы. Мастера Хранилища объектов позволяют создать, например, диалоговое окно, форму для работы с базами данных или отчет. При работе с мастером разработчику в пошаговом режиме предлагается ответить на ряд вопросов, помогающих мастеру создать требуемую форму. Мастера в Windows достаточно часто используются для различных ситуаций, например, при построении диаграмм в пакете Microsoft Office или при установке в компьютер нового оборудования.

Глава 7

Работа с меню

Практически все приложения Windows 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 используется для управления работой всего приложения, каждое из контекстных меню служит для управления отдельным интерфейсным элементом.

Пункт меню представляет собой объект типа `TMenuItem`. Отдельный пункт меню обычно виден как текстовый заголовок, описывающий назначение пункта меню. Пункт меню может быть выделен (маркирован) для указания на включенное состояние. Класс TMenuitem используется для представления пунктов главного и контекстных меню. Основные свойства пункта меню:

- Bitmap типа TBitmap — определяет изображение пиктограммы, размещаемое слева от заголовка пункта меню. По умолчанию свойство имеет значение `nil`, и изображение отсутствует;
- Break типа `TMenuBreak` — задает, разделяется ли меню на колонки. Свойство Break может принимать одно из трех значений:
  - `mbNone` — меню не разделяется (по умолчанию);
  - `mbBreak` — пункты меню, начиная с текущего, образуют новую колонку;
  - mbBreakBar — пункты меню, начиная с текущего, образуют новую колонку, котороая отделена линией;
- `Caption` типа `string` — содержит строку текста, отображаемую 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 — определяет, является ли пункт выделенным. Если свойству установлено значение True, то пункт выделен и в его заголовке появляется специальная отметка. По умолчанию свойство Checked имеет значение False, и пункт меню не имеет отметки;
- AutoCheck типа Boolean — определяет автоматическое изменение значения свойства Checked на противоположное при выборе пользователем пункта меню (в Delphi 6);
- count типа integer — задает количество подпунктов в данном пункте меню. Это свойство есть у каждого пункта меню. Если какой-либо пункт не содержит подпунктов, то свойство Count имеет значение нуль;
- Enabled типа Boolean — определяет, активен ли пункт, т. е. будет ли он реагировать на события от клавиатуры и мыши. Если свойству Enabled установлено значение False, то пункт меню неактивен и его заголовок обесцвечен. По умолчанию свойство Enabled имеет значение True, и пункт меню активен;
- items типа TMenuItems — является массивом подпунктов текущего пункта меню. Каждый пункт меню, имеющий подпункты (вложенное меню), перечисляет их в свойстве items. Это свойство позволяет получить доступ к подпунктам по их позициям в массиве: items [0], items [1] и т. д.;
- RadioItem типа Boolean — определяет вид отметки, появляющейся в заголовке пункта. Если свойству установлено значение False (по умолчанию), то в качестве отметки используется значок в виде галочки, в противном случае пункт отмечается жирной точкой;
- shortcut типа TSortCut — определяет комбинацию 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 через Инспектор объектов, выбрав нужную комбинацию из предлагаемого списка. 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, на которую указывает параметр Key. Если в процессе выполнения программы, например, для пункта меню mnuTest требуется задать комбинацию клавиш <Alt>+<T>, то это можно выполнить следующим образом:

```
mnuTest.ShortCut:= ShortCut(Word('T'), [ssAlt]);
```

- visible типа Boolean — определяет, виден ли пункт на экране. Если свойству visible установлено значение False, то пункт меню на экране не отображается. По умолчанию свойство visible имеет значение True, и пункт виден в меню.

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. Имитация 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 при разработке приложения используется Конструктор меню. Меню также можно создавать или изменять динамически — непосредственно в ходе выполнения приложения.

7.1. Главное меню

Главное меню располагается в верхней части формы под ее заголовком (рис. 7.1) и содержит наиболее общие команды приложения. В Delphi главное меню представлено компонентом MainMenu.

Рис. 7.1. Главное меню

По внешнему виду главное меню представляет собой строку, и его также называют *строчным*. Если пункты меню не умещаются на форме в одну строку, то они автоматически переносятся на следующую строку (рис. 7.2). 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.

При проектировании приложения на форме видны компонент MainMenu и соответствующая ему строка меню. Отображаемая строка меню выглядит

и ведет себя так же, как и при выполнении программы. Для перехода на этапе проектирования приложения в процедуру обработки события onclick пункта меню следует выбрать этот пункт с помощью клавиатуры или мыши.

Рис. 7.2. Главное меню с двумя строками

7.2. Контекстное меню

Контекстное (всплывающее) меню появляется при нажатии правой кнопки мыши и размещении указателя на форме или в области некоторого управляющего элемента. Обычно контекстное меню содержит команды, влияющие только на тот объект, для которого вызвано это меню, поэтому такое меню также называют *локальным*. На рис. 7.3 показан примерный вид контекстного меню.

Рис. 7.3. Контекстное меню

Контекстное меню в Delphi представляется компонентом `PopupMenu`. Его основными свойствами являются следующие:

- `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 появления меню не происходит. 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` — указатель определяет левый верхний край меню (по умолчанию);

- `paCenter` — указатель определяет для меню центр по горизонтали;
- paRight — указатель определяет правый верхний край меню.

Для того чтобы контекстное меню появлялось при щелчке на компоненте, необходимо его свойству PopupMenu присвоить в качестве значения имя требуемого контекстного меню.

Пример. Задание контекстного меню для формы.

```
Form1.PopupMenu := PopupMenu1;
```

Данный оператор задает для формы Forml контекстное меню `PopupMenu1`.

7.3. Конструктор меню

Для создания и изменения меню в процессе разработки приложения в среде Delphi предназначен Конструктор меню (Menu Designer). Запуск Конструктора меню можно выполнить по команде **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 на форму. Напомним, что компоненты MainMenu и PopupMenu размещаются на странице **Standard** (Стандартная) 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. 7.4.

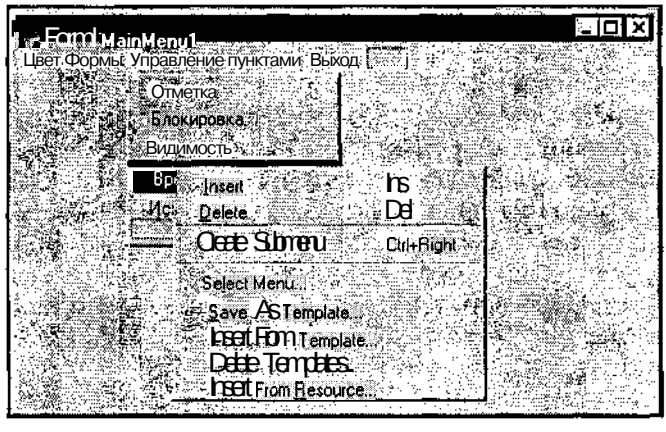


Рис. 7.4. Вид меню при конструировании

При работе с Конструктором меню используются команды его контекстного меню (рис. 7.4), вызываемого щелчком правой кнопкой мыши при разме-

щении указателя в области Конструктора меню. С их помощью можно выполнить такие действия, как добавление **(Insert)** и удаление **(Delete)** пункта меню, создание подменю **(Create Submenu),** выбор меню **(Select Menu).**

При конструировании меню можно также перемещать указателем мыши пункты меню и подменю по технологии drag-and-drop. Используемый совместно с Конструктором меню Инспектор объектов позволяет управлять свойствами отдельных пунктов меню. В частности, наименование пункта меню задается путем присвоения нужного значения его свойству caption.

7.4. Динамическая настройка меню

С помощью Конструктора создание и настройка меню ведется 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;
- переключить форму с одного главного меню на другое.

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. Добавление пункта меню.

```
procedure Form1.mnuItemAddClick(Sender :TSender);
var NewItem :TMenuItem;
begin
  NewItem := TMenuItem.Create(Self);
  NewItem.Caption := 'Новый элемент';
  mnuFile.Add(NewItem);
end;
```

Добавление нового пункта производится в конец списка команд меню **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 на *указанное положение.* Параметр index определяет позицию в массиве элементов меню, на которую вставляется новый пункт. Если значение параметра index выходит за пределы допустимого диапазона, например, больше, чем число подэлементов модифицируемого пункта меню, то возникает исключительная ситуация.

Процедура Delete (index: integer) *удаляет* указанный пункт меню. Если удаляемый пункт имеет подпункты, то они также удаляются.

Пример. Удаление пункта меню.

```
procedure Form1.mnuItemDeleteClick(Sender :TSender);
begin
if mnuFile.Items[2].Caption = 'Второй элемент'
   then mnuFile.Delete(2);
end;
```

Процедура выполняет удаление пункта из подменю **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 *более 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 на форму следует поместить два компонента MainMenu и подготовить соответствующие меню. После этого при выполнении программы возможно подключение к форме любого из этих меню. Для *подключения* к форме главного меню используется СВОЙСТВО Menu формы.

Пример. Переключение между двумя главными меню.

```
if Form1.Menu = EnglishMenu
   then Form1.Menu := RussianMenu
   else Form1.Menu := EnglishMenu;
```

Меню ИМеЮТ названия EnglishMenu И RussianMenu. Код, выполняющий переключение меню, может быть включен в соответствующий обработчик.

Напомним, что Menu является одним из свойств формы, указывающим на главное меню, которое в настоящий момент является *активным.*

Часть II

Работа с базами данных

Глава 8

Введение в базы данных

8.1. Основные понятия

Для успешного функционирования различных организаций требуется наличие развитой информационной системы, которая реализует автоматизированный сбор, обработку и манипулирование данными.

8.1.1. Банки данных

Современной формой информационных систем являются банки данных, включающие в свой состав следующие составляющие:

- вычислительную систему;
- систему управления базами данных (СУБД);
- одну или несколько баз данных (БД);
- набор прикладных программ (приложений БД).

БД обеспечивает хранение информации, а также удобный и быстрый доступ к данным. БД представляет собой совокупность данных различного характера, организованных по определенным правилам.

СУБД представляет собой совокупность языковых и программных средств, предназначенных для создания, ведения и использования БД. По характеру использования СУБД разделяют на персональные и многопользовательские.

Персональная СУБД обеспечивает возможность создания локальных БД, работающих на одном компьютере. К персональным СУБД относятся Paradox, dBase, FoxPro, Access и другие.

Многопользовательские СУБД позволяют создавать информационные системы, функционирующие в архитектуре "клиент-сервер". К много-

пользовательским СУБД относятся Oracle, Informix, SyBase, Microsoft SQL Server, InterBase и др.

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 (Structured Query Language, SQL), обеспечивающий управление структурой БД и манипулирование данными, а также являющийся стандартным средством доступа к удаленным БД;
- • язык запросов по образцу (Query By Example, QBE), обеспечивающий визуальное конструирование запросов к БД.

БД содержит данные, используемые некоторой 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 БД:

- ❒ иерархическую;
- ❒ сетевую;
- ❒ реляционную;
- ❒ объектно-ориентированную.

Большинство современных БД для персональных компьютеров являются реляционными. При последующем изложении материала речь пойдет именно о реляционных 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 БД трактуют в широком смысле слова и обозначают им не только саму БД, но и приложения, обрабатывающие ее данные.

Система Delphi не является СУБД в буквальном смысле этого слова, но обладает возможностями полноценной СУБД. Предоставляемые Delphi средства обеспечивают создание и ведение локальных и клиент-серверных БД, а также разработку приложений для работы практически с любыми БД. Назвать Delphi обычной СУБД мешает, наверное, только то, что, с одной стороны, она не имеет своего формата таблиц (языка описания данных) и 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.

8.1.2. Архитектуры информационных систем

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. Delphi-приложение осуществляет доступ к БД через BDE (Borland Database Engine — процессор баз данных фирмы Borland). BDE представляет собой совокупность динамических библиотек и драйверов, обеспечивающих доступ к данным. Приложение через BDE передает запрос к базе данных, а обратно получает требуемые данные. BDE должен устанавливаться на всех компьютерах, на которых выполняются Delphi-приложения, осуществляющие работу с БД. Отметим, что при установке Delphi на компьютер по умолчанию также устанавливается BDE.

Локальные БД располагаются на том же компьютере, что и работающие с ними приложения. В этом случае информационная система имеет локальную архитектуру (рис. 8.1). 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.

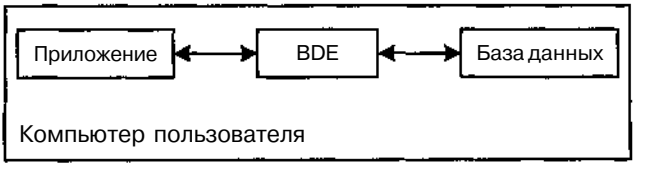


Рис. 8.1. Локальная архитектура

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.

Удаленная БД размещается на компьютере-сервере сети, а приложение, осуществляющее работу с этой БД, находится на компьютере пользователя.

В этом случае речь идет об архитектуре клиент-сервер, когда информационная система делится на неоднородные части — сервер и клиент БД. В связи с тем, что компьютер-сервер находится отдельно от клиента, его также называют *удаленным сервером*.

В архитектуре "клиент-сервер" клиент посылает серверу запрос на предоставление данных и получает только те данные, которые действительно были затребованы. Вся обработка запроса выполняется на удаленном сервере. Достоинствами такой архитектуры являются:

- снижение нагрузки на сеть;
- повышение безопасности информации;
- уменьшение сложности клиентских приложений.

Отметим, что локальные приложения БД называют *одноуровневыми,* а клиент-серверные приложения БД — *многоуровневыми*.

Далее рассматривается работа с локальными БД dBase и Paradox. Отметим, что многие примеры можно также применить и при разработке приложений для работы с удаленными БД.

8.2. Реляционные базы данных

Реляционная БД состоит из взаимосвязанных таблиц. Каждая таблица содержит информацию об объектах одного типа, а совокупность всех таблиц образует единую БД.

8.2.1. 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, например, с помощью Проводника Windows. Однако, в отличие от документов, таблицы БД поддерживают *многопользовательский* режим доступа, т. е. могут одновременно использоваться несколькими приложениями.

Для одной таблицы создается несколько файлов, содержащих данные, индексы, ключи и т. п. Главным из них является файл с данными, имя этого файла является именем таблицы, которое задается при ее создании. В некотором смысле понятие таблицы и ее главного файла являются синонимами, при выборе таблицы выбирается именно ее главный файл: для таблицы dBase им является файл с расширением DBF, а для таблицы Paradox — с расширением DB. Имена остальных файлов таблицы назначаются автоматически — все файлы имеют одинаковое имя, совпадающее с именем таблицы, и разные расширения, указывающие на содержимое соответствующего файла.

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. 8.2). Каждое поле должно иметь уникальное в пределах таблицы имя.

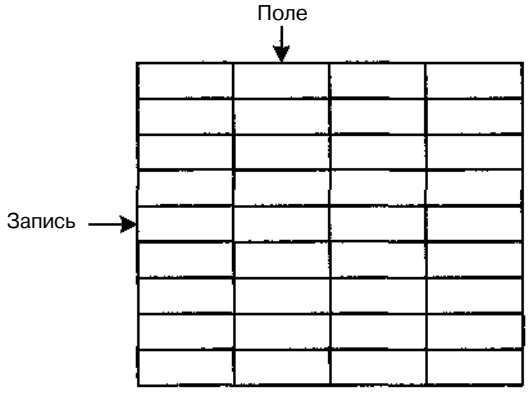


Рис. 8.2. Таблица БД

Поле содержит данные одного из допустимых типов, например, строкового, целочисленного или даты. 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 исключительная ситуация.

Особенности организации таблиц зависят от конкретной СУБД, используемой для создания и ведения БД. Например, в таблице dBase нет поля автоинкрементного типа (с автоматически наращиваемым значением) и в ней нельзя определить ключ. Подобные особенности необходимо учитывать при выборе типа (формата) таблицы, так как 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.

Основу таблицы составляет описание ее полей, каждая таблица должна иметь хотя бы одно поле. Понятие структуры таблицы является более широким и включает в себя:

- 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. 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 названия и характеристики полей, состав и названия ключа и индексов, ограничения. Однако название таблицы и ее файлов остается прежним.

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 с помощью любой программы (в том числе и с помощью Проводника Windows).

8.2.2. 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;
- предотвращение повторения значений ключа;
- ускорение выполнения запросов к БД;
- установление связи между отдельными таблицами БД;
- использование ограничений ссылочной целостности.

Ключ также называют *первичным ключом* или *первичным (главньш) 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 строковых полей большого размера 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, состоящие в следующем.

- 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, например, 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 является использование для него поля соответствующего типа, которое автоматически обеспечивает поддержку уникальности значений. Для таблиц Paradox таким является поле автоинкрементного типа, еще одно достоинство которого заключается в небольшом размере (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 том 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 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 (поиска) к данным;
- ❒ сортировку записей;
- установление связи между отдельными таблицами БД;
- использование ограничений ссылочной целостности.

В двух последних случаях индекс используется 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. Индекс служит для сортировки таблиц по индексным полям. 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. Так, для компонента Table сортировка записей автоматически выполняется на основании значений именно текущего индекса.

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).

8.2.3. Способы доступа к данным

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, то его также называют SQL-ориентированным. Этот способ доступа ориентирован на выполнение операций с удаленными БД и является предпочтительным при работе с такими БД, хотя его можно использовать также 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, для реализации навигационного способа доступа можно использовать компоненты `Table` или Query, для реляционного — компонент Query.

8.2.4. Связь между таблицами

В частном случае БД может состоять из одной таблицы, например, с днями рождения сотрудников организации. Однако обычно реляционная БД состоит из 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 подчиненной таблице для связи с главной таблицей берется 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. 8.3 показана схема связи между таблицами Paradox.

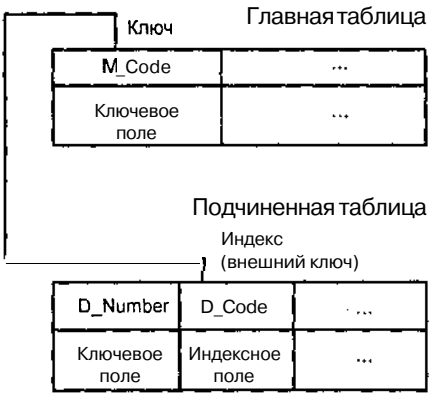


Рис. 8.3. Схема связи между таблицами Paradox

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 буквой `M` (Master), а названия полей подчиненной таблицы начинаются буквой 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".

Наиболее часто используется отношение "один-ко-многим", которое означает, что одной записи главной таблицы в подчиненной таблице может соответствовать несколько записей, в частном случае ни одной. После установления связи между таблицами при перемещении в главной таблице 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.

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.

Работа со связанными таблицами имеет следующие особенности.

- При изменении (редактировании) связанного поля 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 ограниче-

ния, 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 существуют, то необходимо удалить их или, наоборот, запретить удаление записей из обеих таблиц. И в том, и в другом случае пользователю должно быть выдано предупреждение.

8.2.5. 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, утверждаются. 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 возникновении сбоя, 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 затем 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.

8.2.6. 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. Отметим, что здесь бизнес-правила являются 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.

8.2.7. Форматы таблиц

Delphi не имеет своего формата таблиц, но поддерживает, как свои 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 и совместимость с большим числом приложений.

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.

Благодаря своим достоинствам таблицы Paradox используются чаще. В табл. 8.1 содержится список типов полей для таблиц Paradox 7. Для каждого типа приводится символ, используемый для обозначения этого типа в программе Database Desktop, и описание значений, которые может содержать поле рассматриваемого типа.

Таблица 8.1. *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, TIF, GIF и EPS. При загрузке в поле изображение преобразуется к формату BMP. Для хранения изображения используется файл с расширением MB

Таблица 8.1 (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 |, так как 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.

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.

8.3. Средства для работы с базами данных

Хотя Delphi не имеет своего формата таблиц БД, она, тем не менее, обеспечивает мощную поддержку большого количества различных СУБД Средства Delphi, предназначенные для работы с БД, можно разделить на два вида:

- инструментальные средства;
- компоненты.

К инструментальным средствам относятся специальные программы и пакеты, обеспечивающие обслуживание БД вне разрабатываемых приложений.

Компоненты предназначены для создания приложений, осуществляющих операции с БД.

8.3.1. Инструментальные средства

Для операций с БД система Delphi предлагает набор инструментальных средств, перечисленных ниже.

- 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.
- 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.
- Data Pump — программа для переноса данных между БД.

8.3.2. Компоненты

Кроме компонентов, Delphi также предоставляет разработчику специальные объекты, например, объекты типа `TField`. Как и другие управляющие элементы 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.

Основная часть компонентов, используемых для работы с БД, находится на страницах **Data Access, Data Controls, BDE** и **QReport** Палитры компонентов.

Замечание

Состав страниц Палитры компонентов, а также состав расположенных на них компонентов различаются в зависимости от версии Delphi. Ниже приведены страницы Палитры компонентов, соответствующие Delphi 6. В предыдущих версиях страница **BDE** отсутствует, а большинство ее компонентов, включая `Table` и `Query`, расположено на странице **Data Access.**

На странице **Data Access** (рис. 8.4) находятся невизуальные компоненты, предназначенные для организации доступа к данным:

- `DataSource` - источник данных;
- `ClientDataSet` - клиентский набор данных;
- DataSetProvider - провайдер набора данных.

Рис. 8.4. Страница **Data Access**

На странице **Data Controls** (рис. 8.5) находятся визуальные компоненты, предназначенные для управления данными:

- DBGrid — сетка (таблица);
- DBNavigator — навигационный интерфейс;
- DBText — надпись;
- DBEdit — однострочный редактор;
- `DBMemo` — многострочный редактор;
- `DBImage` — графический образ;
- DBListBox — простой список;
- `DBComboBox` — комбинированный список;
- `DBCheckBox` — независимый переключатель;
- `DBRadioGroup` — группа зависимых переключателей;
- `DBLookupListBox` — простой список, формируемый по полю другого набора данных;
- `DBLookupComboBox` — комбинированный список, формируемый по полю другого набора данных;
- `DBRichEdit` — полнофункциональный тестовый редактор;
- `DBCtrlGrid` — модифицированная сетка;
- DBChart — диаграмма.

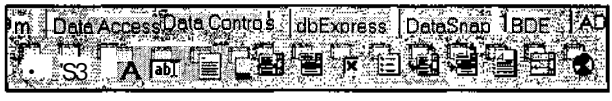


Рис. 8.5. Страница Data Controls

На странице **BDE** (рис. 8.6) находятся 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 — модификация набора данных, основанного на SQL-запросе;

- NestedTable — **вложенная таблица;**
- BDEClientDataSet — **клиентский набор данных.**

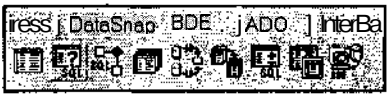


Рис. 8.6. Страница **BDE**

На странице QReport (рис. 8.7) находятся компоненты (в основном визуальные), предназначенные для построения отчетов:

- QuickRep — **отчет;**
- QRSubDetail — **полоса отчета для таблиц, связанных отношением "главный-подчиненный";**
- QRStringsBand — **строковая полоса отчета;**
- QRBand — **полоса отчета;**
- QRChildBand — **дочерняя полоса отчета;**
- QRGroup — **группа;**
- QRLabel — **надпись;**
- QRDBText — **текстовое поле набора данных;**
- QRExpr — **выражение;**
- QRSysData — **системная информация;**
- QRMemo — **многострочный текст;**
- QRExprMemo — **многострочное выражение;**
- QRRichText — **форматированный текст;**
- QRDBRichText — **форматированный текст поля набора данных;**
- QRShape — **геометрическая фигура;**
- QRImage — **графический образ;**
- QRDBImage — **графический образ поля набора данных;**
- QRCompositeReport — **составной отчет;**
- QRPreview — **окно просмотра отчета;**
- QRTextFilter — **текстовый фильтр;**
- QRCSVFilter — **CSV-фильтр;**
- QRHTMLFilter — **HTNL-фильтр;**
- QRChart — **диаграмма.**

Рис. 8.7. Страница **QReport**

Названия многих компонентов, предназначенных для работы с данными, содержат префиксы, например, DB ИЛИ QR. Префикс `DB` означает, что визуальный компонент связан с данными и используется для построения интерфейсной части приложения. Такие компоненты размешаются на форме и предназначены для управления данными со стороны пользователя. Префикс QR означает, что компонент используется для построения отчетов. Эти компоненты размещаются на компоненте `QuickRep` отчета и его элементах, например, на полосе QRBand, и служат для оформления внешнего вида отчета.

8.4. Технология создания информационной системы

В качестве первого примера использования возможностей Delphi по работе с БД рассмотрим технологию создания простой информационной системы. Эту информационную систему можно создать без программирования, она не требует написаний кода: все необходимые операции выполняются с помощью программы Database Desktop, Конструктора формы и Инспектора объектов. При создании информационной системы основными являются следующие этапы:

- создание БД;
- создание приложения.

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 Program Files\Shared Files\Borland Shared\Data. Готовые таблицы можно использовать также и для своих приложений.

8.5. Создание таблиц базы данных

Для работы с таблицами БД при проектировании приложения удобно использовать программу Database Desktop, которая позволяет выполнять следующие действия:

- создание таблицы;
- изменение структуры;
- редактирование записей.

Кроме того, с помощью Database Desktop можно выполнять и другие действия над БД (создание, редактирование и выполнение визуальных запросов и SQL-запросов, операции с псевдонимами).

Процесс создания новой таблицы начинается по команде **File | New | Table** (Файл | Создать | 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. 8.8) 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.

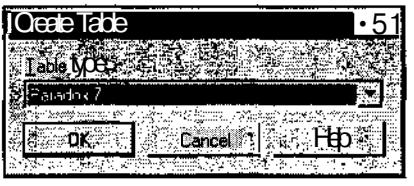


Рис. 8.8. Выбор формата таблицы

После выбора формата таблицы появляется окно определения структуры таблицы (рис. 8.9), 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 для таблиц определенных форматов, например, для таблиц Paradox.

После определения структуры таблицы ее необходимо сохранить, нажав кнопку **Save as** (Сохранить как) 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.

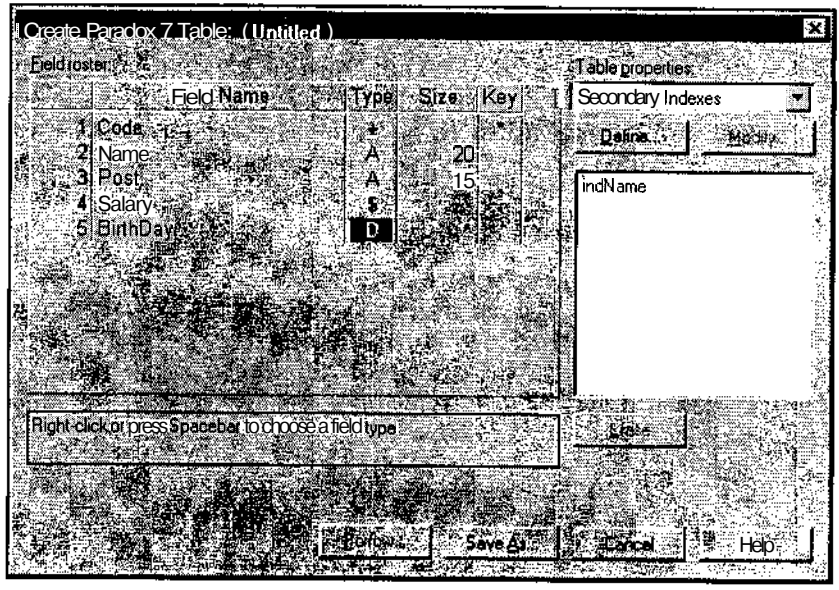


Рис. 8.9. Определение структуры таблицы

8.5.1. Описание полей

Центральной частью окна определения структуры таблицы является список **Field roster** (Список полей), в котором указываются поля таблицы. Для каждого поля задаются:

- имя — в столбце **Field Name;**
- тип — в столбце **Type**;
- размер — в столбце **Size.**

Имя поля вводится по правилам, установленным для выбранного формата таблиц. Правила именования и допустимые типы полей таблиц Paradox описаны во введении в БД.

Тип поля можно задать, непосредственно указав соответствующий символ, например, A для символьного или I для целочисленного поля, или выбрать его из списка, раскрываемого при нажатии клавиши <Пробел> или щелчком правой кнопки мыши в столбце **Type** (Тип). 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** (Тип).

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 стро-

кового 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 описанием полей 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 о 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. 8.9).

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), содержащий следующие пункты:

- **Secondary Indexes** — 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** (Индексы) и **Table Language** (Язык таблицы).

8.5.2. Задание индексов

Задание индекса сводится к определению:

- состава полей;
- параметров;
- имени.

Эти элементы устанавливаются или изменяются при выполнении операций создания, изменения и удаления индекса.

Для выполнения операций, связанных с заданием индексов, необходимо выбрать пункт **Secondary Indexes** (Вторичные индексы) комбинированного списка, 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. 8.9 это индекс **indName**.

Напомним, что для таблиц Paradox индекс также называют вторичным индексом.

Создание нового индекса начинается с нажатия кнопки **Define** (Определить), являющейся всегда доступной. Это приводит к появлению окна **Define Secondary Index** (Задание вторичного индекса), в котором задаются состав полей и параметры индекса (рис. 8.10).

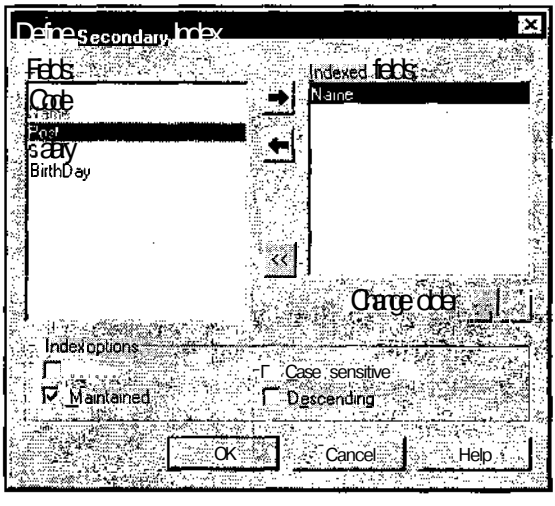


Рис. 8.10. Окно задания индекса

В списке **Fields** (Поля) 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 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** (Изменение порядка). Для перемещения поля (полей) необходимо его (их) выделить и нажать нужную кнопку.

Переключатели, расположенные в нижней части окна задания индекса, позволяют указать следующие параметры индекса:

- **Unique** — индекс требует для составляющих его полей уникальных значений;
- **Maintained** — если таблица открыта, индекс автоматически не модифицируется;
- **Case sensitive** — для полей строкового типа учитывается регистр символов;
- **Descending** — сортировка выполняется в порядке убывания значений.

Так как для таблиц dBase нет ключей, то для них использование параметра **Unique** (Уникальный) 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** (Сохранить индекс как), в котором указывается имя индекса (рис. 8.11). 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, так как для таблиц Paradox подобная система именования используется при автоматическом образовании имен для обозначения ссылочной целостности между таблицами. После повторного нажатия OK сформированный индекс добавляется к таблице, и его имя появляется в списке индексов.

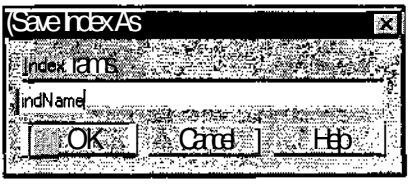


Рис. 8.11. Задание имени индекса

Созданный индекс можно изменить, определив новый состав полей, параметров и имени индекса. Изменение индекса не отличается от процесса его создания. После выделении индекса в списке и нажатия кнопки Modify (Изменить) снова открывается окно задания индекса (см. рис. 8.10). При нажатии кнопки **OK** появляется окно сохранения индекса (рис. 8.11), содержащее имя изменяемого индекса, которое можно исправить или оставить прежним.

Для удаления индекса его нужно выделить в списке индексов и нажать кнопку **Erase** (Удалить). В результате индекс удаляется без предупреждающих сообщений.

Кнопки **Modify** (Изменить) и **Erase** (Удалить) доступны, только если индекс выбран в списке.

8.5.3. 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 систем типа Database Desktop, так и для приложений, создаваемых в Delphi. Дополнительно к этим ограничениям или вместо них в приложении можно также задать программные ограничения.

Для выполнения операций, связанных с заданием ограничений на значения полей, нужно выбрать пункт **Validity Checks** (Проверка значений) комбинированного списка **Table properties** (Свойства таблицы) (см. рис. 8.9).

8.5.4. Задание ссылочной целостности

Понятие ссылочной целостности относится к связанным таблицам и проявляется в следующих вариантах взаимодействия таблиц:

- запрещается изменение поля связи или удаление записи 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** (Справочная целостность) комбинированного списка **Table properties** (Свойства таблицы) (см. рис. 8.9).

8.5.5. Задание паролей

Пароль позволяет задать права доступа пользователей (приложений) к таблице. Если для таблицы задать 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** (Защита паролем) в комбинированном списке **Table properties** (Свойства таблицы) окна определения структуры таблицы (см. рис. 8.9).

8.5.6. Задание языкового драйвера

Для задания языкового драйвера нужно выбрать пункт **Table Language** (Язык таблицы) комбинированного списка **Table properties** (Свойства таблицы) окна определения структуры таблицы (см. рис. 8.9).

8.5.7. Изменение структуры таблицы

Структуру существующей таблицы можно изменить, выполнив команду **Table | Restructure** (Таблица | Изменение структуры) 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... (Сохранить как) и указать 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 фай-

лов, поэтому простое переименование всех файлов таблицы приведет к ошибке при попытке обратиться к таблице.

Если необходимо просто ознакомиться со структурой таблицы, то выполняется команда **Table | Into Structure** (Таблица | Структура). 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.

8.5.8. Работа с псевдонимами

Для работы с псевдонимами БД можно использовать инструментальное средство **Alias Manager** (Менеджер псевдонимов) (рис. 8.12), вызываемый по команде **Tools | Alias Manager** (Сервис | Менеджер псевдонимов) меню программы Database Desktop.

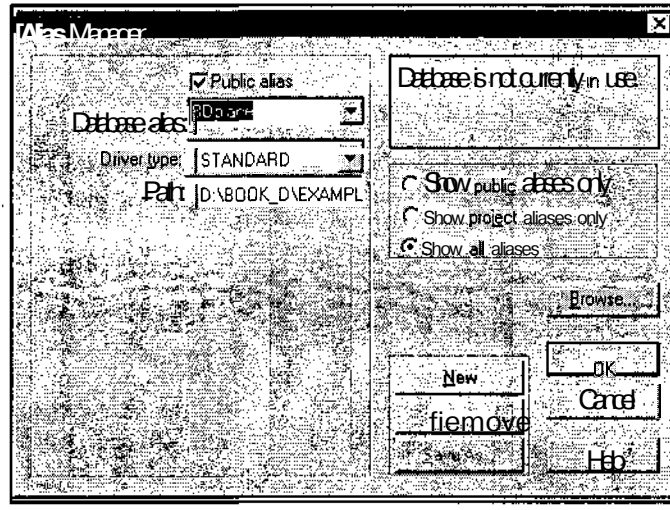


Рис. 8.12. Окно Менеджера псевдонимов

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** (Псевдонимы БД). 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, вызываемом нажатием кнопки Browse (Обзор).

Информация о псевдонимах сохраняется в конфигурационном файле idapi.cfg процессора баз данных.

8.6. Создание приложения

Для примера рассмотрим создание приложения, 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. 8.13, где на форме размещены компоненты Table1, DataSource1, DBGrid1 и DBNavigator1.

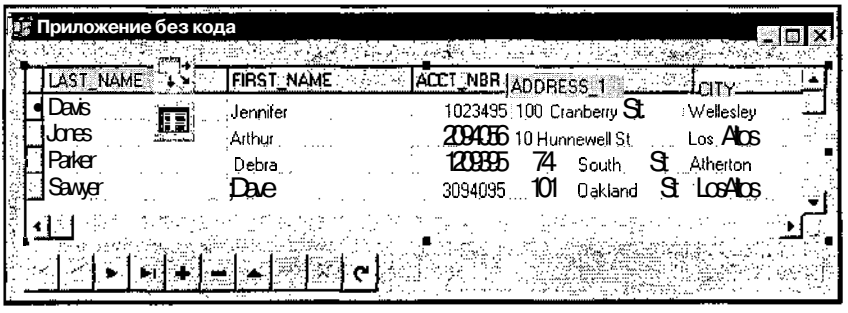


Рис. 8.13. Форма приложения для работы с БД

Компонент Table1 обеспечивает взаимодействие с таблицей БД. Для связи с требуемой таблицей нужно установить соответствующее значение свойствам `DataBaseName`, указывающему путь к БД, и `TableName`, указывающему имя таблицы. После задания таблицы для открытия набора данных свойству Active должно быть установлено значение True.

Замечание

Значение True **свойству** Active **нужно устанавливать после задания таблицы БД, то есть после установки нужных значений свойств** DataBaseName **и** TableName.

Имя таблицы лучше выбирать из раскрывающегося списка 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.

Компонент DataSourcel является промежуточным звеном между компонентом Table1, соединенным с реальной таблицей БД, и визуальными компонентами DBGrid1 и DBNavigatorl, с помощью которых пользователь взаимодействует с этой таблицей. На компонент Table1, с которым связан компонент DataSourcel, указывает свойство DataSet последнего.
Компонент DBGridl 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 DBNavigatorl позволяет пользователю осуществлять перемещение по таблице, редактировать, вставлять и удалять записи. Компоненты DBGridl и DBNavigatorl связываются со своим источником данных — компонентом DataSourcel — через свойства DataSource.

Взаимосвязь компонентов приложения и таблицы БД и используемые при этом свойства компонентов показаны на рис. 8.14.

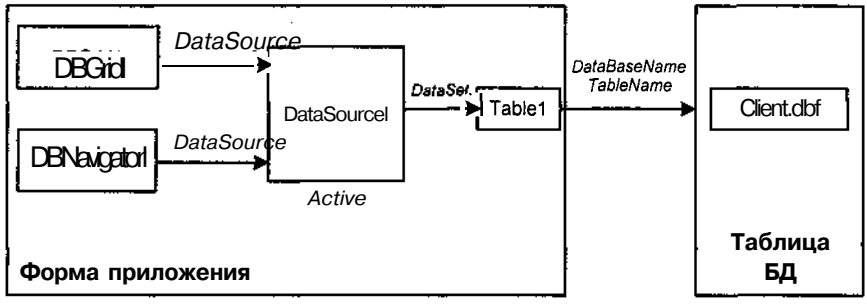


Рис. 8.14. Взаимосвязь компонентов приложения и таблицы БД

При разработке приложения значения всех свойств компонентов можно задать с помощью Инспектора объектов. При этом требуемые значения можно набрать в поле значений или выбрать из раскрывающихся списков. В последнем случае приложение создается с помощью мыши и не требует набора каких-либо символов с клавиатуры. В табл. 8.2 приведены компоненты, используемые для работы с таблицей БД, основные свойства и их значения.

В дальнейшем при организации приложений предполагается, что названные компоненты связаны между собой именно таким образом, и свойства, с помощью которых эта связь осуществляется, не рассматриваются.

Для автоматизации процесса создания формы, использующей компоненты для операций с БД, можно вызвать **Database Form Wizard** (Мастер форм баз данных). Этот Мастер расположен на странице **Business** Хранилища объектов.

Мастер позволяет создавать формы для работы с отдельной таблицей, а также со связанными таблицами, при этом можно использовать наборы данных Table ИЛИ Query.

Таблица 8.2. Значения свойств компонентов

Компонент	Свойства	Значения
Table1	DataBaseName	dbdemos
	TableName	clients.dbf
	Active	true
DataSource1	DataSet	Table1
DBGrid1	DataSource	DataSource1
DBNavigator1	DataSource	DataSource1

Глава 9

Компоненты для работы с данными

9.1. Компоненты доступа к данным

Компоненты доступа к данным являются невизуальными. В этом разделе рассматриваются основные компоненты доступа к данным.

9.1.1. 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 служат такие компоненты, как Table И Query, КОТОрые ЯВЛЯЮТСЯ ПРОИЗВОДНЫМИ ОТ класса TDBDataSet — потомка класса TDataSet (через класс TBDEDataSet). Они имеют схожие с базовыми классами характеристики и поведение, но каждый из них имеет

и свои особенности. Рассмотрим наиболее общие характеристики наборов 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 (alias), то его можно выбрать через Инспектор объектов в раскрывающемся списке.

Замечание

Желательно задавать имя БД через псевдоним. Это заметно облегчает перенос приложения и файлов БД в другие каталоги и на другие компьютеры, так как для обеспечения работоспособности приложения после изменения расположения БД достаточно изменить название каталога, на который ссылается псевдоним БД. Псевдоним можно создать с помощью программ Database Desktop или BDE Administrator.

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 свойству Active значение False. В противном случае генерируется исключительная ситуация.

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).

Пример. Перебор всех записей набора данных.

```
var i: integer;
...
Table1.First;
for i := 1 to Table1.RecordCount do begin
    // Здесь можно расположить операторы, 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, начиная 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 сортировки 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.

Пример. Переход к записи с указанным номером.

```
Table1.RecNo := StrToInt(Edit1.Text);
```

Здесь выполняется переход к записи, номер которой содержится в поле редактирования Editl. Эта 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.

9.1.2. 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 открытие набора данных невозможно, то при попытке установить свойству Active значение True выдается сообщение об ошибке, а свойство Active сохраняет значение False. Одной из причин невозможности открытия набора данных может быть неправильное значение СВОЙСТВ TableName ИЛИ SQL.

Замечание

На этапе проектирования свойству Active наборов данных автоматически устанавливается значение False при изменении значения свойств DataBaseName, TableName или SQL.

Пример. Управление состоянием набора данных.

Рассмотрим 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 свойству Active значения True. Процедура `Close` закрывает набор данных, ее вызов эквивалентен установке свойству Active значения False.

При открытии набора данных любым способом возникают события BeforeOpen И AfterOpen типа `TDataSetNotifyEvent`, который описывается следующим образом:

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
```

В этом описании параметр `DataSet` определяет набор данных, для которого произошло событие.

При закрытии набора данных возникают события BeforeClose И `AfterClose` типа 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 необходимо 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.

Пример. Сохранение изменений при закрытии набора данных.

```
procedure TForm1.Table1BeforeClose(DataSet: TDataSet);
begin
if (Table1.State = dsEdit) or (Table1.State = dsInsert) then Table1.Post;
end;
```

В приведенном примере, 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.

9.1.3. Режимы наборов данных

Наборы данных могут находиться в различных режимах. Текущий режим набора данных определяется свойством state типа TDataSetstate, которое доступно для чтения во время выполнения приложения. 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).

Выделим следующие основные режимы набора данных.

- dsInactive — неактивность; набор данных закрыт и доступ к его данным невозможен. В этот режим набор данных переходит после его закрытия, когда свойству Active установлено значение False.
- dsBrowse — осуществляется навигация по записям набора данных и просмотр данных. В этот режим набор данных переходит следующим образом:
  - из режима dsInactive — при установке свойству Active значения True;
  - **ИЗ режима dsEdit — При вызове методов** Post **ИЛИ** Cancel;
  - **ИЗ режима** dsInsert — **при ВЫЗОВе** методов 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 данный режим набор данных переходит ИЗ режима dsBrowse при вызове методов Insert, InsertRecord, **Append ИЛИ AppendRecord.**
- dsCalcFields — расчет вычисляемых полей. Используется обработчик события OnCalcFields.
- 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 ттаЫе, так как для компонента Query отбор записей осуществляется средствами языка SQL.
- 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 OnFiiterRecord набор данных автоматически переводится в режим dsBrowse.

Взаимосвязи между основными режимами наборов данных показаны на рис. 9.1, где приведены также некоторые методы и свойства, с помощью которых набор данных переходит из одного режима в другой.

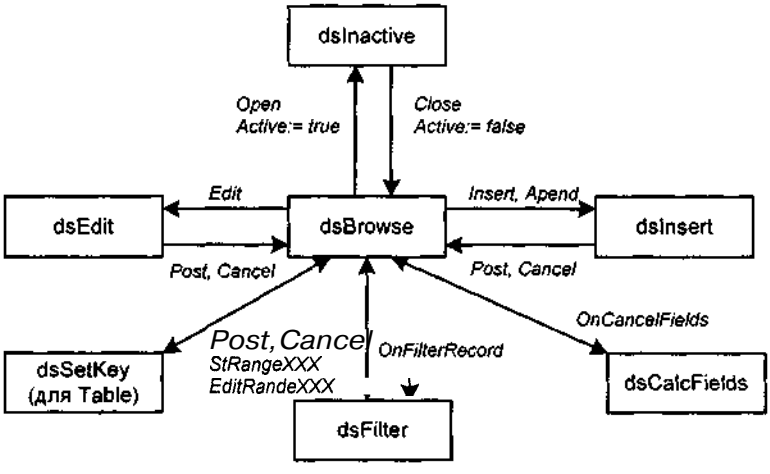


Рис. 9.1. Взаимосвязи режимов наборов данных

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.

При выполнении программы определить режим набора данных можно 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.

Пример. Анализ режима набора данных.

```
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;
```

Здесь 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 DataSourcel.

9.1.4. 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 набор данных имеет соответствующие методы и свойства, доступные при выполнении приложения.

Для доступа к полям удобно использовать метод FieldByName. Функция FindField(const FieldName: String): TFieid **Возвращает ДЛЯ набора данных** поле, имя которого указывает параметр FieldName. Если заданное параметром FieldName поле не найдено, то генерируется исключительная ситуация.

> **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.

Метод FieidByName часто используется для доступа к значению поля текущей записи совместно с такими свойствами объекта Field, как AsString, AsInteger, AsFloat или AsBoolean, которые соответственно позволяют обращаться к значению поля как к строковому, целочисленному, вещественному или логическому значению.

Пример. Чтение содержимого полей текущей записи.

```
Var x: integer;
...
Label1.Caption := Table1.FieldByName('Name').AsString;
x := Table1.FieldByName('Number').AsInteger;
```

Здесь 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 исключительная ситуация.

9.1.5. Особенности набора данных *Table*

Компонент Table представляет собой набор данных, который в некоторый момент времени может быть связан только с *одной* таблицей БД. Этот набор данных формируется на основе 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.

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 значения свойству TabieName 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 свойству Active набора данных автоматически устанавливается значение False. При задании имени таблицы программным способом набор данных предварительно необходимо закрыть, установив его свойству Active значение False. В противном случае генерируется исключительная ситуация.

Пример. Задание имени 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 TabieName. Набор данных Table1 предварительно

закрывается и снова открывается уже после смены таблицы. Тип таблицы определяется автоматически по расширению имени файла.

Свойство TaBleType типа TTtabieType определяет тип таблицы. Для локальных таблиц это свойство может принимать следующие значения:

- ttDefauit — тип таблицы автоматически определяется по расширению файла;
- ttParadox — таблица Paradox ;
- ttDBase — таблица dBASE;
- ttFoxPro — таблица FoxPro;
- ttASCII — текстовый файл, содержащий данные в табличном виде (ASCII-таблица).

Если свойство TableType имеет значение ttDefauit (по умолчанию), то тип таблицы определяется по расширению файла:

- DB или отсутствует — таблица Paradox;
- DBF — таблица dBASE;
- TXT — текстовый файл (ASCII-таблица).

По умолчанию в состав набора данных Table попадают все записи связанной с ним таблицы. Для отбора записей, удовлетворяющих определенным условиям, используются фильтры.

Чтобы запретить пользователям изменять содержание записей, можно использовать свойство Readonly типа Boolean. По умолчанию оно имеет значение False, что предоставляет пользователю право на модификацию записей.

В наборе данных Table возможно указание текущего индекса, требуемого для выполнения операций:

- сортировки записей;
- поиска записей;
- установки связей между таблицами.

Текущий индекс устанавливается с помощью свойства IndexName или IndexFieldNames типа string. На этапе разработки приложения текущий индекс выбирается из списка 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 indexFieidNames является имя *поля,* для которого был создан индекс. Если используется индекс, состоящий 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.

Пример. Задание текущего индекса.

```
Table1.IndexName := 'indName';
Table2.IndexFieldNames := 'Name';
```

Здесь компоненты Table1 и таЫе2 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 indexFieidNames, перечислив ключевые поля таблицы, так как ключ не имеет имени и поэтому недоступен через СВОЙСТВО IndexName.

Задать ключ в качестве текущего индекса можно так:

```
Table1.IndexFieldNames := 'Name;Post;BirthDay';
```

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 IndexFieidNames взаимозависимы. При установке значения одного из них другое автоматически очищается.

Индекс, устанавливаемый текущим, должен существовать. Если индекс, задаваемый как значение свойства IndexName или IndexFieidNames, для таблицы не существует, то возникает исключительная ситуация.

При смене таблицы, с которой ассоциирован компонент Table, значения свойств IndexName и IndexFieidNames не изменяются автоматически, поэтому программист должен самостоятельно установить нужные значения.

Получить доступ к полям в составе текущего индекса можно с помощью СВОЙСТВ IndexFieldCount И IndexFields.

9.1.6. 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. 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. 9.2), вызываемый через Инспектор объектов двойным щелчком в области значения свойства SQL.

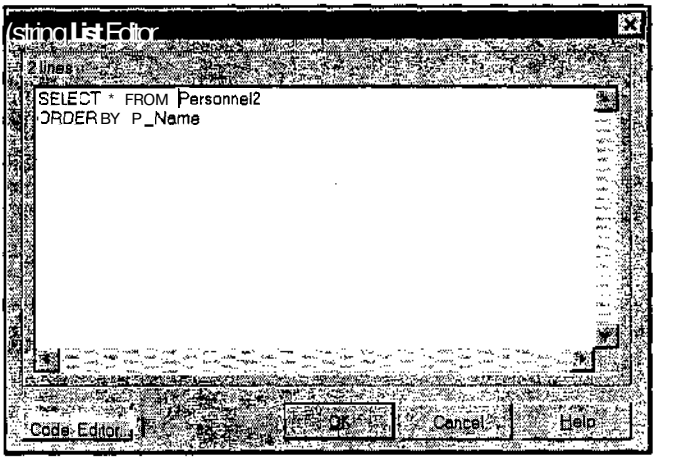


Рис. 9.2. Редактирование запроса SQL

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.

Пример. Создадим приложение — простейший редактор, позволяющий подготавливать и выполнять SQL-запросы.

На рис. 9.3 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.

Редактирование SQL-запроса осуществляется с помощью компонента `Memo1`. Набранный запрос выполняется при нажатии кнопки `Button1` с надписью **Выполнить,** а результат выполнения отображается в компоненте DBGridi.

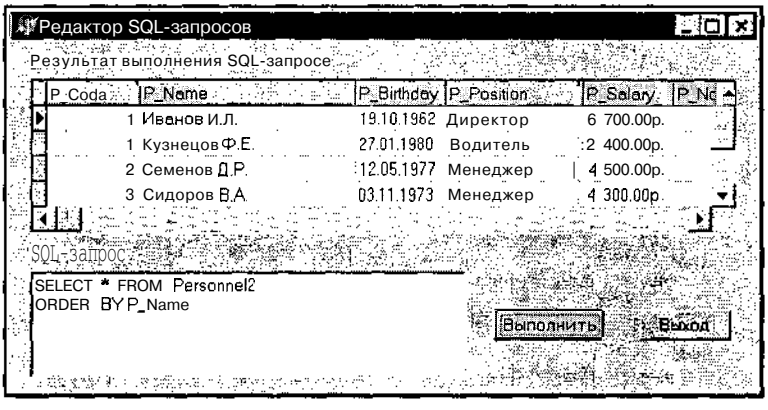


Рис. 9.3. Форма приложения-редактора SQL-запросов

При наличии в тексте SQL-запроса ошибки генерируется исключительная ситуация и выдается сообщение об ошибке (рис. 9.4), а результат запроса оказывается не определенным. При этом набор данных `Query1` автоматически закрывается.

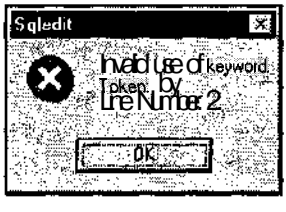


Рис. 9.4. Сообщение об ошибке

Значения СВОЙСТВ DataSet ИСТОЧНИКОВ данных DataSourcel И `DataSource` сетки DBGridl, с помощью которых организуется взаимодействие компонентов Queryl, DataSourcel и DBGridl, устанавливаются при создании формы. В последующих примерах приложений значения этих свойств устанавливаются через Инспектор объектов, поэтому операторы, присваивающие свойствам требуемые значения, в модуле формы отсутствуют.

Приведем код модуля uSQLEdit формы `Form1` приложения:

```
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);
 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 принадлежит свойство SQL компонента Query1 и свойство 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 СВОЙСТВУ Active значения True.

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 свойству RequestLive нужно установить 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 свойству RequestLive значения True игнорируется.

Чтобы проверить результат установки значения свойству 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 СВОЙСТВУ RequestLive ЗНачеНИЯ True, должны быть выполнены определенные условия, в частности, данные должны отбираться только из одной таблицы, и эта таблица должна допускать модификацию.

Вместо компонента Table можно также использовать компонент Query. Если установить свойству SQL значение SELECT * FROM NameTableBD, а свойству RequestLive — значение True, то набор данных Query будет аналогичен набору данных Table (здесь NameTabieBD является именем таблицы БД, которое для компонента Table задается в свойстве TableName). Однако, в отличие от Table, набор данных Query не имеет системы индексов, поэтому к нему не применимы методы, использующие индексацию, например, методы FindFirst, FindLast, FindNext И FindPrior.

9.1.7. Объекты поля

Объекты типа TField являются невизуальными и служат для доступа к данным соответствующих полей записей набора данных.

Объект поля Field имеет тип TField и является полем набора данных. Тип `TField` является абстрактным классом и непосредственно не используется. Вместо него применяются производные классы, 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, с 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). В табл. 8.1 приведены основные типы объектов Field.

Таблица 9.1. *Основные типы объектов Field*

Тип объекта	Вид поля
`TBLOLBField`	**BLOB-поле**
`TMemoField`	**Memo-поле (поле комментария)**
`TGraphicField`	**Графическое поле**
`TDateTimeField`	**Поле даты и времени**
`TNumericField`	**Числовое поле**
`TBCDField`	**Поле BCD-значения**
`TFloatField`	**Поле вещественного значения**
`TCurrencyField`	**Поле значения денежной суммы**
`TintegerFieid`	**Поле целочисленного значения**
`TAutoincField`	**Поле автоинкрементного значения**
`TStringFieid`	**Поле строкового значения**

Существуют следующие два способа задания состава полей набора данных:

- 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 использование динамических полей имеет 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.

Отметим, что при модификации структуры таблицы, например, удалении поля или изменении его типа, открытие набора данных, имеющего статические поля, может привести к возникновению исключительной ситуации.

Динамические и статические поля имеют одинаковые свойства, события и методы, с помощью которых можно управлять этими объектами при выполнении приложения. Статические поля определяются на этапе разработки приложения, поэтому многие их свойства доступны через Инспектор объектов.

9.1.8. 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 — считаться 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.

Для запуска Редактора полей (рис. 9.5) следует дважды щелкнуть на компоненте Table или Query или вызвать щелчком правой кнопкой мыши для этих компонентов контекстное меню и выбрать пункт **Fields Editor.** Большую часть Редактора занимает список статических полей, при этом поля перечисляются в порядке их создания, который может отличаться от порядка следования полей в таблице БД.

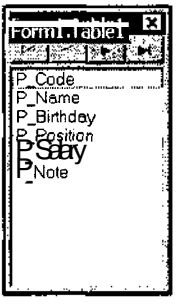


Рис. 9.5. Редактор полей

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 возможно задание или изменение свойств этого поля (объекта типа TField) и определение обработчиков его событий. Подобные действия возможны потому, что соответствующие статическим полям объекты типа TField доступны на этапе разработки приложения.

Для создания статического поля следует вызвать контекстное меню Редактора полей и выбрать пункт **Add Fields** (Добавить поля).

Для удаления статического поля нужно выбрать пункт Delete контекстного меню или выделить в списке поле и нажать клавишу <Delete>. После удаления статического поля оно становится недоступным для операций в программе, но при необходимости его снова можно сделать статическим, добавив в список Редактора полей. При этом все свойства этого поля устанавливаются заново, а все сделанные ранее изменения теряются.

Замечание

Если удалены все статические поля, то тогда 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 попей 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.

9.1.9. 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 нужно путать свойства Name и FieldName, они обозначают разные объекты и в общем случае могут не совпадать.

Пример. Обращение к полю в программе.

```
Table1.FieldByName('Number').DisplayLabel := 'Количество';
Table1Number.DisplayLabel := 'Количество';
```

Для статического поля Number возможны два способа обращения: по имени поля в наборе данных и по имени объекта Field поля.

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.

Пример. Доступ к значению поля 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;
```

Здесь чтение значения поля `Salary` выполняется 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 и 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 исключительную ситуацию.

Если поле является невидимым (свойству `visible` установлено значение False), но разрешено для редактирования (свойству Readonly установлено значение False), то возможно изменение значения этого поля программно.

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 LoadFromFiie.

9.1.10. 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.

Для анализа режима, в котором находится набор данных, можно использовать свойство state типа TDatasetstate. При каждом изменении режима 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 свойству AutoEdit значения False.

Замечание

Значение свойства AutoEdit влияет на возможность редактирования набора данных только со стороны пользователя. Программно можно изменять записи независимо от значения этого свойства.

Независимо от 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, который описан следующим образом:

```
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 за положением указателя текущей записи и выполнения действий, связанных с его перемещением. Это событие генерируется также при открытии набора данных.

Вот как осуществляется контроль за перемещением 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.

9.2. Визуальные компоненты

Визуальные компоненты для работы с данными расположены на странице **Data Controls** (Элементы управления данными) 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 графический Образ 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** (Стандартная) и **Additional** (Дополнительная)

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 содержимое определенного поля текущей записи набора данных.

Отметим, что для всех визуальных компонентов, предназначенных для отображения и редактирования полей, при изменении пользователем их содержимого набор данных автоматически переводится в режим редактирования. Произведенные с помощью этих компонентов изменения 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 автоматически в режим редактирования не переводится. 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.

Отметим, что определенная часть компонентов, используемых для формирования отчетов (страница **QReport** (Быстрый отчет) Палитры компонентов), тоже имеет свои аналоги среди других визуальных компонентов. В табл. 9.2 приводятся так называемые стандартные и дополнительные визуальные компоненты, расположенные на страницах **Standard** (Стандартная) и **Additional** (Дополнительная) Палитры компонентов, а также соответствующие им визуальные компоненты для работы с данными (страница **Data Controls** (Элементы управления данными)) и для формирования отчетов (страница **Qreport** (Быстрый отчет)).

Рассмотрим компоненты: сетку DBGrid и навигационный интерфейс DBNavigator, которые не имеют аналогов среди компонентов, рассмотренных ранее.

Таблица 9.2. *Соответствие визуальных компонентов, расположенных на разных страницах Палитры компонентов*

Компоненты страниц Standard и Additional	Компоненты страницы Data Controls	Компоненты страницы QReport
Label	DBText	QRLabel
Edit	DBEdit	_

Таблица 9.2. (окончание)

Компоненты страниц Standard и Additional	Компоненты страницы Data Controls	Компоненты страницы QReport
Memo	DBMemo	—
RichEdit	DBRichEdit	QRRichEdit, QRDBRichEdit
ListBox	DBListBox	—
ComboBox	DBComboBox	—
CheckBox	DBCheckBox	—
RadioGroup	DBRadioGroup	—
Image	DBImage	QRImage, QRDBImage
Shape	—	QRShape
StringGrid	DBGrid	—
Chart	DBChart	QRChart

9.2.1. 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 отображаются в ней. 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 ее поля. Вставка записи происходит в месте нахождения указателя текущей записи. Изменения, сделанные при редактировании или добавлении записи, подтверждаются нажатием клавиши <Enter> или переходом к другой записи, а отменяются нажатием клавиши <Esc>. Для удаления записи следует нажать комбинацию клавиш <Ctrl>+<Delete>.

9.2.2. Характеристики сетки

Несмотря на то, что по своему виду сетка DBGrid похожа на сетку StringGrid, между ними имеются значительные различия. Так, у сетки StringGrid можно устанавливать через соответствующие свойства число ее строк и столбцов. У сетки DBGrid числом строк управлять нельзя, так как 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 **доступно во время выполнения программы и 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 от 0 до** 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** FixedCoior — **значение** clBtnFace **(цвет КНОПКИ).**

Свойство TitleFont **типа** TFont **определяет шрифт, используемый для вывода заголовков столбцов.**

Доступ к параметрам сетки 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** dgAiwaysShowEditor;
- 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 свойству Options устанавливается значение [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 следующим образом:

```
type TDBGridciickEvent = procedure (Column: TColumn) of object;
```

Параметр column представляет 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 потере его.

Сетка DBGrid способна автоматически отображать в своих ячейках информацию, при необходимости программист может выполнить и собственное отображение сетки. Например, когда желательно выделить ячейку или столбец с помощью цвета или шрифта, а также вывести в ячейке кроме текстовой, также и графическую информацию, в частности, небольшой рисунок. Для программного отображения сетки используется обработчик события OnDrawColumnCell ТИПа TDrawColumnCellEvent, возникающего При прорисовке любой ячейки. Тип события OnDrawColumnCell описан следующим образом:

```
type  TDrawColumnCellEvent = procedure (Sender: TObject;
       const Rect: TRect; DataCol: Integer;
       Column: TColumn; State: TGridDrawState) of object;
```

Параметр Rect содержит координаты ограничивающего ячейку прямоугольника, параметр DataCoi определяет номер прорисовываемой колонки 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 DefauitDrawing имеет зна-

чение False, то вызывается событие OnDrawColumnCell, в обработчике которого следует разместить операции по прорисовке области сетки.

Пример. Программная прорисовка сетки.

```
// Свойству DefaultDrawing должно быть установлено 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 ImageListl (третий по счету). Список с рисунками можно подготовить заранее при разработке приложения и загрузить динамически в процессе выполнения программы.

При прорисовке ячеек используется свойство Canvas элемента DBGrid1, а также параметр Rect. Если обработчик события OnDrawColumnCell является общим для нескольких сеток DBGrid, то при отображении их ячеек вместо названия конкретного компонента (в примере DBGridi) нужно задать КОНСТРУКЦИЮ (Sender as TDBGrid) ИЛИ TDBGrid (Sender). Например, оператор

```
(Sender as TDBGrid).Canvas.Font.Color := clRed;
```

устанавливает красный цвет для символов ячеек сетки, указываемой параметром Sender.

9.2.3. Столбцы сетки

Отдельный столбец сетки представляется объектом Column 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 определяется полями набора данных (а также соответствующими объектами типа TField), для которых создаются объекты типа TCoiumn.

Функционирование динамических столбцов зависит от свойств объекта поля: при изменении свойств объекта типа TFieid соответственно изменяются свойства объекта типа TCoiumn. К примеру, динамический столбец получает от поля такие характеристики, как название 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 название, то оно не будет меняться даже в случае, если с этим столбцом связывается другое поле набора данных. Кроме того, объекты типа TCoiumn статических столбцов создаются на этапе разработки приложения, и их свойства доступны через Инспектор объектов.

Для запуска Редактора столбцов (рис. 9.6) можно вызвать контекстное меню компонента DBGrid и выбрать в нем пункт **Columns Editor** (Редактор столбцов) или выполнить щелчок мышью на свойстве Columns в Инспекторе объектов.

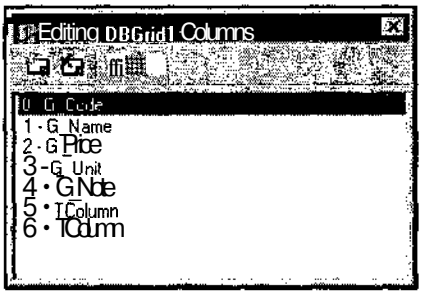


Рис. 9.6. Окно Редактора столбцов

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** (Панель инструментов) контекстного меню Редактора столбцов. Большую часть Редактора столбцов занимает список статических столбцов, в нем столбцы перечисляются в порядке их создания (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, это означает, 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 возможно задание или изменение его свойств и определение обработчиков его событий. Это допустимо потому, что соответствующие статическим столбцам объекты типа TColumn доступны уже на этапе разработки приложения.

Вновь создаваемые статические столбцы получают значения свойств по умолчанию, зависящие также от полей набора данных, с которыми эти столбцы связаны.

Объект столбца доступен через свойство Columns типа `TDBGridColumns`. При проектировании приложения свойства этого объекта (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. С помощью Инспектора объектов этому свойству значение можно задавать путем выбора из списка.
- 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 на которую 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 и возможностью модификации значений, не отличаются от соответствующих свойств поля.

Пример. Установка свойств для столбцов сетки.

В наборе данных определено пять полей, для каждого из которых с помощью Редактора столбцов создан статический столбец. Текст процедуры, выполняемой при создании формы приложения, имеет следующий вид:

```
// Значения свойств можно установить в Инспекторе 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;
```

Здесь первый и третий столбец 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 можно проделать с помощью Инспектора объектов. Вид формы при выполнении приложения показан на рис. 9.7.

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.

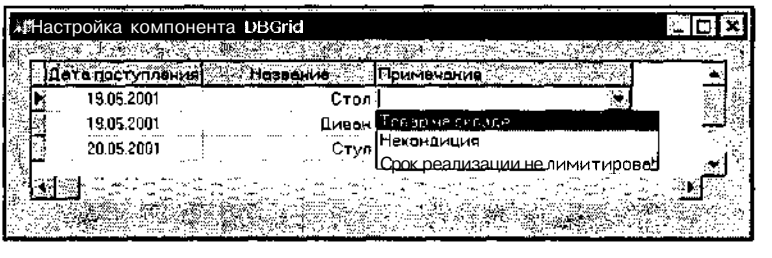


Рис. 9.7. Установка свойств для столбцов сетки

Пример. Преобразование значений записей набора данных в текст.

В качестве набора данных используется компонент `Query1`, для которого SQL-запрос вводится в многострочное поле редактирования `Memo1`. Выполнение запроса происходит при нажатии кнопки `Button1` с названием **Выполнить SQL.** Полученные в результате выполнения запроса записи отображаются в сетке `DBGrid1`. При нажатии кнопки Button2 с названием **Преобразовать** происходит последовательный просмотр полей всех записей набора данных (сетки) и преобразование их в текст, помещаемый в многострочное поле редактирования `Memo2` (рис. 9.8).

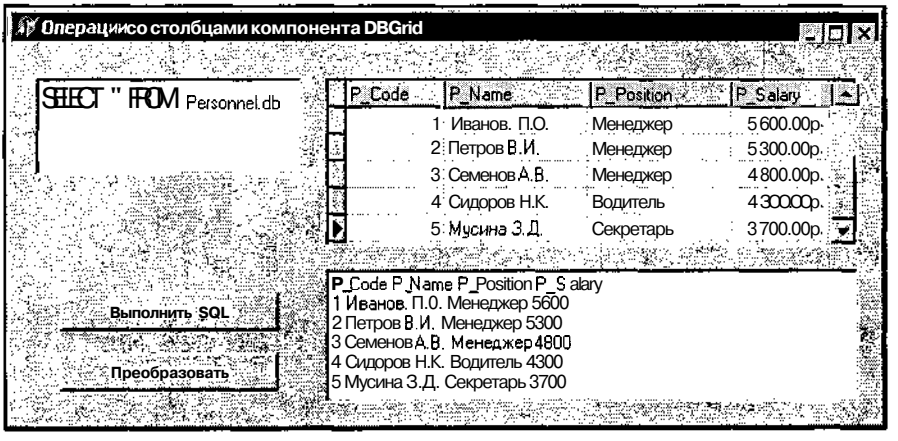


Рис. 9.8. Преобразование записей набора данных в текст

Ниже приведены обработчики событий нажатия кнопок.

```
// 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;
```

```
/I 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.

9.2.4. 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. 9.9).

Рис. 9.9. 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. Например, строку кода:

```
DBNavigator.BtnClick(nbNext);
```

имитирует нажатие кнопки nbNext, вызывающей переход к следующей записи набора данных.

При нажатии кнопки Delete Record (Удалить запись) 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 этому свойству значение False, то запись будет удаляться без подтверждения.

Свойство Flat типа Boolean управляет *внешним видом* кнопок. По умолчанию оно имеет значение False, и кнопки отображаются в объемном виде. При установке свойству Flat значения 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.

Пример. Использование отдельных кнопок для вызова методов управления набором данных.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.Next;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Table1.Prior;
end;
```

Здесь при нажатии кнопок Button1 и Button2 выполняется переход к следующей и предыдущей записям набора данных Table1, соответственно.

Глава 10

Операции с данными

Операции с данными рассматриваются на примере использования навигационного способа доступа к локальным БД. При этом можно использовать наборы данных Table ИЛИ Query.

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:

- сортировка записей;
- навигация по набору данных;
- фильтрация записей;
- редактирование записей;
- вставка и удаление записей.

Отметим, что аналогичные операции применимы к набору данных и при реляционном способе доступа, реализуемом с помощью SQL-запросов.

10.1. 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.

Сортировка заключается в упорядочивании записей по определенному полю в порядке возрастания или убывания содержащихся в нем значений.

Сортировку можно выполнить и по нескольким полям. Например, при сортировке по двум полям записи сначала 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. Далее рассматривается сортировка набора данных Table, в то время как для компонента Query сортировка выполняется средствами языка SQL.

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 для индекса признак ixDescending 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 IndexFieidNames.

Пример. Сортировка 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.

Пример. Сортировка 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).

Пример. Управление сортировкой.

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 `btnSort` с надписью **Сортировать.** Вид формы при проектировании показан на рис. 10.1.

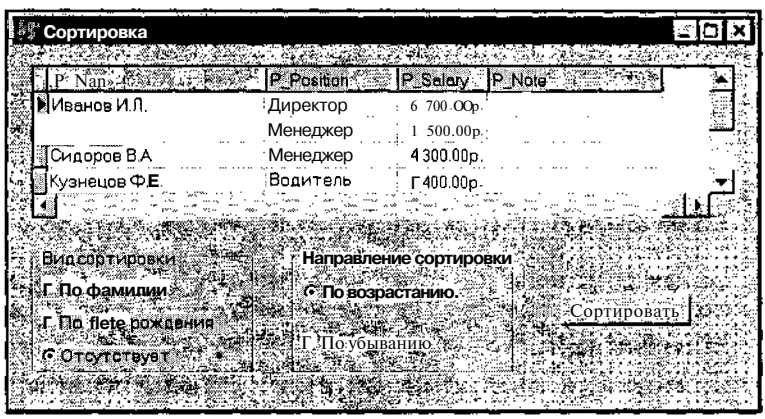


***Рис.* 10.1.** Вид формы для сортировки набора данных

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.

Ниже приводится обработчик события нажатия кнопки btnsort, вызывающей 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.

10.2. 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.

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 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.

> **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.

При любом изменении положения указателя текущей записи для набора данных генерируются события `BeforeScroll` и `AfterScroll` типа `TDataSetNotifyEvent`.

Пример. Управление указателем текущей записи.

```
procedure TForm1.Button3Click(Sender: TObject);
var sum: real;
    n:   integer;
begin
sum := 0;
// Установка текущего указателя на первую запись
Table1.First;
for n := 1 to Table1.RecordCount do begin
    inc(sum, Table1.FieldByName('Salary').AsFloat);
    // Перемещение текущего указателя на следующую запись
    Table1.Next;
    end;
Label2.Caption := FloatToStr(sum);
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, который имеет свойство RecordCount.

Аналогичным образом можно перебрать все записи набора данных, начиная с последней. Естественно, при этом нужно использовать методы Last И Prior.

Для *контроля за положением* 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 свойству RecNo значения, равного 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, находится 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 значение 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.

Пример. Работа с указателем текущей записи.

```
procedure TForm1.Button2Click(Sender: TObject);
var sum: real;
begin
sum := 0;
Table1.First;
while not Table1.EOF do begin
      sum := sum + Table1.FieldByName('Salary').AsFloat;
      Table1.Next;
      end;
Label2.Caption := FloatToStr(sum);
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.

При перемещении по записям набора данных связанные с ним визуальные компоненты отображают изменения данных, причем смена отображения может происходить достаточно быстро, вызывая мерцание 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 используются методы DisableControls И EnableControls.

Пользователь имеет возможность *перемещаться* по набору данных с помощью управляющих элементов, 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 кнопкок **First record, Prior record, Next record** или **Last record** компонента DBNavigatorl 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 DBNavigatorl.

Другим вариантом является размещение на форме управляющих элементов, например, кнопки Button, предназначенных для навигации по набору данных. При нажатии кнопки вызывается соответствующий метод перемещения текущего указателя в заданном направлении. Например, при нажатии кнопки с названием **Предыдущая запись** вызывается метод Prior. Кроме

того, часто на форме также размещаются элементы (обычно кнопки) для управления такими операциями, как редактирование, вставка записей, фильтрация и сортировка набора данных.

10.3. Фильтрация записей

Фильтрация — это задание ограничений для записей, отбираемых в набор данных. 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 похожа на SQL-запросы, но является менее мощным средством. По сравнению с SQL-запросами фильтрация менее эффективна, так как ограничивает количество записей, видимых в наборе.

Система Delphi дает возможность осуществлять фильтрации записей:

- по выражению;
- по диапазону.

Как более универсальную рассмотрим фильтрацию по выражению, при использовании которой набор данных ограничивается записями, удовлетворяющими выражению фильтра, задающему условия отбора записей. Важным достоинством фильтрации по выражению является то, что она применима к любым полям, в том числе к неиндексированным.

Для задания выражения фильтра используется свойство Filter типа string. Выражение фильтра 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 выражение фильтра 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 используется свойство Filter типа Boolean. По умолчанию это свойство имеет значение False, и фильтрация выключена. При установке свойству Filtered значения 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 исключительная ситуация. Если фильтр не активен (свойство Filtered имеет значение False), то выражение фильтра на корректность не анализируется.

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 рекомендуется отключать, чтобы различать слова, написанные в различных регистрах;
- foNoPartiaicompare — 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 foNoPartiaicompare. Например, при выключенном значении foNoPartiaicompare для фильтра Post = 'в*' 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 [].

Пример. Обработчики событий формы, используемой для фильтрации записей набора данных по выражению.

Вид формы приведен на рис. 10.2. Управление фильтрацией набора данных выполняется с помощью двух кнопок и поля редактирования.

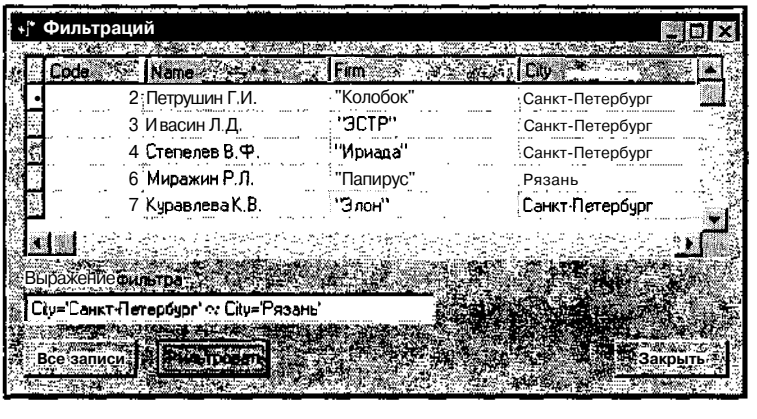


Рис. 10.2. Фильтрация по выражению

При нажатии кнопки btnFilter с надписью **Фильтровать** 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 btnAllRecord с надписью **Все записи** фильтр отключается, при этом показываются все записи.

Ниже 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 независимые (checkBox) и зависимые (RadioButton) переключатели.

Пример. Управление фильтрацией по выражению с использованием шаблонов.

Рассмотрим обработчики событий формы приложения, в которой пользователю предоставлена возможность управлять фильтрацией по двум полям или по выражению либо совсем отключать фильтрацию (рис. 10.3)

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

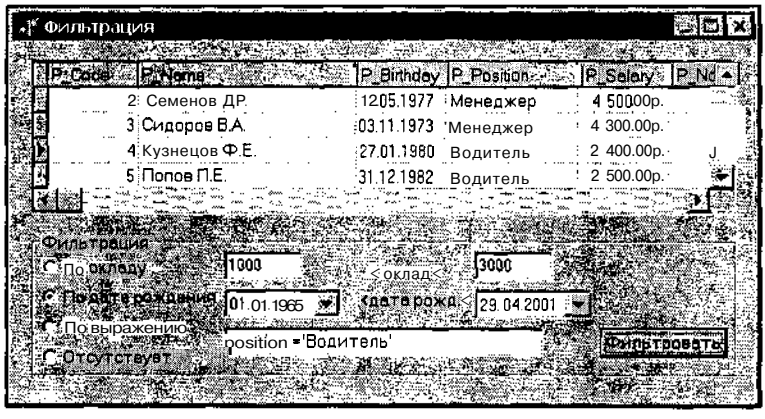


Рис. 10.3. Управление фильтрацией по выражению

этом неравенстве минимальное и максимальное значения. Фильтрация записей происходит при нажатии кнопки **Фильтровать. В** обработчике события нажатия этой кнопки на основании выбранного для фильтрации поля и введенных пользователем значений происходит автоматическое формирование выражения фильтра.

Ниже приведены обработчики событий модуля формы Formi 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 :=
      'P_Salary > ' + edtSalaryMin.Text +
      ' AND P_Salary < ' + edtSalaryMax.Text;
// Фильтровать по дате рождения
if rbFilterBirthday.Checked then Table1.Filter :=
      'P_BirthDay > '+ DateToStr(dtpBirthdayMin.Date) +
      ' AND P_Birthday < ' + DateToStr(dtpBirthdayMax.Date);
// Фильтровать по введенному 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 свойства 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.

10.4. Поиск записей

Поиск заключается в нахождении записей, данные которых удовлетворяют определенным условиям. 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.

10.4.1. 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.

Если имя поля или тип значения заданы неправильно, то при попытке выполнить метод Locate генерируется исключительная ситуация.

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 loPartiaiKey и foNoPartialCompare имеют противоположное действие: первое из них допускает, а второе запрещает частичное совпадение значений.

Замечание

При наличии у параметра Options значения loPartiaiKey к нему автоматически добавляется значение locaseinsensitive.

Пример. Поиск 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 Variant, в котором содержится несколько значений. Для приведения к типу вариантного массива используется функция VarArrayOf. Значения должны разделяться запятыми и быть заключены в квадратные скобки, порядок значений должен соответствовать порядку полей параметра KeyFields. Например:

```
Table1.Locate('Name;Post;', VarArrayOf(['П', 'Инженер']),
 [locaseinsensitive, loPartiaiKey]);
```

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 управляющих элементов, расположенных на форме. При этом действия пользователя по управлению поиском в наборе данных мало, чем отличаются от аналогичных действий при выполнении фильтрации.

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 отличие между двумя методами заключается в том, что метод Lookup осуществляет поиск на *тонное* 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.

10.4.2. 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.

10.5. Модификация набора данных

Модификация 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 с помощью свойства Readonly типа Boolean, при присвоении которому значения True изменения записей запрещаются. 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 управляющих элементов, например, редакторов Edit и Memo, запрет на редактирование относится как к визуальному (пользователем), так и к программному изменению записей набора данных.

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 исключительную ситуацию.

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 свойству AutoEdit источника данных DateSource для визуальных компонентов должно быть установлено значение True (по умолчанию). Если этому свойству установить 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 влияния на другие управляющие элементы, такие как, например, кнопка Button или переключатель checkBox.

При модификации набора данных для связанного с ним источника данных DateSource генерируется событие OnUpdateData.

После модификации набора данных возможна ситуация, когда сделанные изменения не отображаются визуальными компонентами, связанными с этим набором данных. В таких случаях нужно вызывать метод Refresh, который повторно считывает набор данных. Вызов этого метода гарантирует, что визуальные компоненты будут отображать текущие, а не устаревшие данные.

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.

Пример. 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 на форме Form2. Оба набора связаны с одной и той же физической таблицей. В модуле формы `Form1` должна быть ссылка на модуль формы Form2.

10.5.1. 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 по поиску и перемещению на требуемую запись. После того, как указатель текущей записи установлен на нужную запись и набор данных находится в режиме просмотра, для редактирования записи следует:

- перевести набор данных в режим редактирования;
- изменить значения полей записи;
- 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 исключительная ситуация;
- если набор данных уже находился в режиме редактирования или вставки, то никаких действий не происходит;
- если набор данных пуст, то он переходит в режим вставки.

Перед вызовом метода Edit можно 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 на 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 нужно дважды щелкнуть 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 управляющих компонентов, *не связанных* с набором данных, например, кнопок Button или переключателей CheckBox, программист должен самостоятельно кодировать действия по предотвращению попыток редактирования немодифицируемого набора данных.

Пример. Блокировка редактирования немодифицируемого набора данных.

```
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 названия **Редактировать** или **Edit.** Перед переводом в этот режим выполняется проверка, можно ли изменять записи набора данных Table1, и если нет, то процедура выдает соответствующее сообщение и завершается.

Другой способ предотвратить редактирование — блокирование управляющих элементов, выполнение обработчиков событий которых в настоящий момент времени невозможно.

Пример. Процедура с блокированием управляющих элементов.

```
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 приведенном коде переключатель cbEditBan с возможным заголовком **Редактирование запрещено** указывает, допустимо ли изменять записи набора данных Table1. Если этот переключатель установлен, то модификация набора данных запрещается, при этом также блокируется кнопка btnEdit вызова метода Edit.

Отметим, что 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 возбудить "тихое" исключение.

При переходе в режим редактирования с помощью кнопки Button проверку его допустимости можно выполнить в обработчике события ее нажатия. Однако использование обработчика события BeforeEdit обычно удобнее, так как оно генерируется при переводе набора данных в режим редактирования любым способом.

Пример. Процедура — обработчик события BeforeEdit.

```
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 операторов 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 операторов набор данных Table1 должен находиться в режиме редактирования или вставки. Если данные в редакторах Edit2 или Edit3 содержат данные в формате, не соответствующем целому и вещественному числам, то генерируется исключительная ситуация.

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 исключительной ситуации. 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.

Пример. Процедура, 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(Editl.Text);
   except
   MessageDlg('Неправильный коэффициент!' + #13#10 +
              'Повторите ввод.', mtError, [mbOK], 0);
   if Editl.CanFocus then Edit1.SetFocus;
   exit;
   end;
// Запоминание позиции текущего указателя
bml := 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 отображения 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 изменяются на коэффициент, который введен в редактор **Editl**.

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 закреплены. Поэтому в приведенном примере метод `Post` можно было не вызывать, так как сразу после него вызывается метод Next. Однако при использовании методов FindFirst, FindLast, FindNext И FindPrior незакрепленные 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 закрепляются при переходе к другой записи или нажатии клавиши <Enter>, а в компоненте DBNavigator можно нажать кнопку **Post Edit** (Утвердить 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 режим, который был 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 управляющие элементы 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 в нескольких записях можно обратиться к методу `RollBack` класса TDateBase.

При редактировании текущей записи последовательность операторов присваивания и вызовов метода Post можно заменить вызовом метода SetFields (const `Values`: array of const), который устанавливает все или часть значений полей текущей записи.

10.5.2. Добавление записей

Добавлять записи можно только к модифицируемому набору данных, в противном случае будет сгенерирована исключительная ситуация.

Для добавления записи нужно выполнить следующие действия:

- перевести набор данных в режим вставки;
- задать значения полей новой записи;
- 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 необходимо 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 операторы присваивания или 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.

Пример. Добавление новой записи.

```
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. После выполнения этого метода набор данных автоматически возвращается в режим просмотра, и запись считается включенной в набор данных.

Пример. Установка значений вновь добавленной записи методом SetFields.

```
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(const Values: array of const)` **объединяет функ-**циональность методов insert и SetFields, вставляя 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 управляющих элементов формы. Для компонента DBGrid новая запись добавляется к набору данных при нажатии клавиши <Insert> или при переходе на последнюю запись. Если на 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. Например:

```
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 генеруются события BeforePost и AfterPost или BeforeCancel И AfterCancel.

10.5.3. 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.

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 исключительную ситуацию.

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 исключительная ситуация, и генерируется событие OnDeleteError, в обработчике которого можно выполнить собственный анализ ошибки.

Удаление нескольких последовательно расположенных записей имеет особенность, связанную с тем, что при вызове метода Delete в цикле по пере-

бору 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;
```

Здесь 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.

10.6. 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-ко-многим", когда одной записи в главной таблице может соответствовать несколько записей в подчиненной таблице. Такая связь также называется "мастер-детальный". 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.

10.6.1. 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. 10.1 и 10.2. В название полей включены префиксы S и C (ПО первым буквам названий таблиц). Такое обозначение помогает при установке связи между таблицами — из названия поля сразу видно, к какой таблице оно принадлежит.

Таблица 10.1. *Структура таблицы store*

Имя поля	Обозначение типа	Размер	Ключевое поле	Примечание
S_Code	+		*	Уникальный код товара. Используется для связи с подчиненной таблицей
S_Name	A	20		Название товара. Требует обязательного заполнения

***Таблица** 10. 1(окончание)*

Имяполя	Обозначение типа	Размер	Ключевое поле	Примечание
S_Unit	A	7		Единица измерения. Требует обязательного заполнения
S_Price	$			Цена единицы товара. Требует обязательного заполнения
S_Quantit У	N			Количество товара на складе
S_Note	A	30		Примечание

***Таблица** 10.2. Структура таблицы Cards*

Имя поля	Обозначение типа	Размер	Ключевое поле	Примечание
C_Number	+		*	Уникальный код записи о движении товара
C_Code	I			Код записи о движении товара, используемый для связи с главной таблицей
C_Move	N			Приходное или расходное количество. Требует обязательного заполнения
C_Date	D			Дата прихода или расхода

Между таблицами устанавливается связь "главный-подчиненный", при которой таблица store склада является главной, а таблица Cards движения товара — подчиненной (рис. 10.4). 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.

Приложение для работы со складом включает главную форму fmStore (рис. 10.5) и форму fminput ввода данных о новом товаре.

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 те записи, которые соответствуют 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.

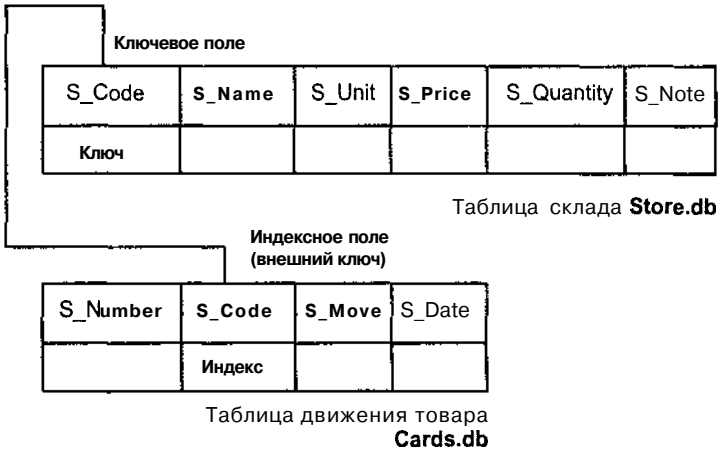


Рис. 10.4. Связь между таблицами `Store` и `Cards`

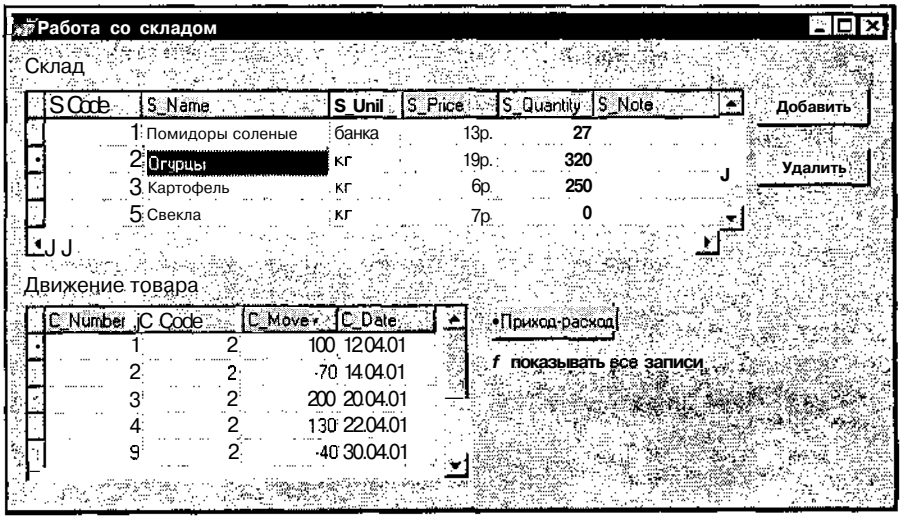


Рис. 10.5. Окно приложения Работа со складом

Модификация данных таблиц с помощью компонентов DBGrid запрещена, для этого их свойствам AutoEdit установлено значение False. Для модификации таблиц используются кнопки Button, а также отдельная форма `fmInput`. Обработчики событий нажатия кнопок btnNew с названием **Добавить** и `btnDelete` с названием **Удалить** добавляют записи о новом товаре в таблицу склада и удаляют записи о товаре.

При нажатии кнопки btnNew выводится в модальном режиме форма fminput (рис. 10.6), 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. 10.6. 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 нажата. Если кнопка OK, то сделанные изменения принимаются, в противном случае — нет.

Для удаления записи с данными о товаре следует нажать кнопку btnDeiete, 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 `btnMove` с надписью **Приход-расход.** 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 заносится текущая дата с помощью оператора присваивания.

Пользователь должен ввести 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 выполняется 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 sQuantity количества товара в таблице склада.

Для разрыва связи между таблицами используется переключатель cbMoveAii с надписью **Показывать все записи.** По умолчанию он выключен, 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, так как 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 их редактировать с помощью компонентов `DBGrid`, а в коде модулей эти поля не затрагиваются.

Ниже приведены коды модулей форм приложения. Установка свойств большинства компонентов выполнена в обработчиках событий создания форм приложения.

```
// 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);
  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.IndexFieldNames := 'C_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);
  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 при выполнении приложения. Обычно 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 MasterFieids в Инспекторе объектов. В списке **Available Indexes** (Доступные индексы) выбирается индекс подчиненной таблицы, после чего составляющие его поля отображаются в списке **Detail Fields** (Детальное поле). В этом списке необходимо выбрать поле подчиненной таблицы, а в списке **Master Fields** (Главное поле) — поле главной таблицы (рис. 10.7). После нажатия кнопки **Add** (Добавить) выбранные поля связываются между собой, что отображается в списке **Joined Fields** (Связанные поля), например, следующим образом `C_Code -> scode`. При этом оба поля пропадают из своих списков. Заполнение свойств `IndexName` и MasterFieids происходит после закрытия окна при нажатии кнопки OK.

Замечание

Перед вызовом окна Редактора полей связи необходимо установить значение свойства MasterSource подчиненного набора данных TableCard, которое должно указывать на источник данных `dsstore` главной таблицы.

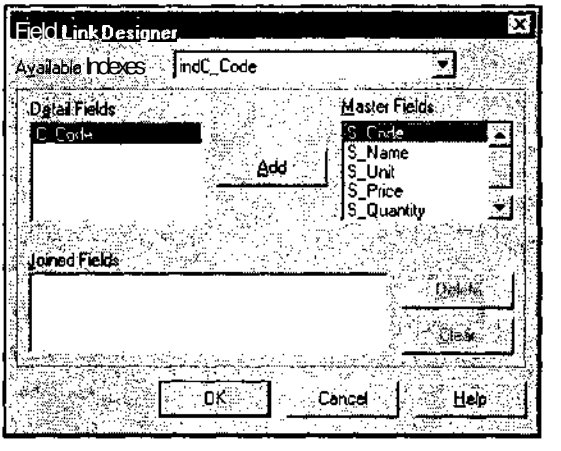


Рис. 10.7. Редактор полей связи

Аналогичным образом можно установить связь одной таблицы с несколькими таблицами, например, таблицей приходной накладной и таблицами поставщиков, покупателей и товаров.

10.6.2. Использование механизма транзакций

При работе с несколькими таблицами БД взаимосвязанные изменения периодически вносятся в разные таблицы. Например, в рассмотренном выше примере добавлялись новые записи о приходе или расходе товара в таблицу движения товара и соответственным 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.

Для реализации механизма явных транзакций Delphi предоставляет специальные методы StartTransaction, Commit И Rollback компонента DataBase. Метод StartTransaction начинает транзакцию, после него долж-

ны располагаться операторы, составляющие транзакцию. При выполнении операций выполняется обработка возникающих исключений. Если исключений не возникло, то после выполнения всех операций вызывается метод Commit, утверждающий 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.

Расширим приведенное выше приложение по работе со складом за счет включения в него механизма транзакций, изменив код обработчика события нажатия кнопки btnMove так, как показано ниже. Изменения касаются вставки записи в таблицу TableCard и редактирования записи таблицы TableStore.

Для вызова методов, связанных с запуском и завершением транзакции, на 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 операторов SQL также можно управлять транзакциями.

Глава 11

Подготовка отчетов

Отчет — это печатный документ, содержащий данные, аналогичные получаемым в результате выполнения запроса к БД. В Delphi для создания отчетов служит генератор отчетов QuickReport, содержащий обширный набор компонентов.

11.1. Компоненты отчета

Компоненты, предназначенные для создания отчетов, находятся на странице **QReport** (Быстрый отчет) Палитры компонентов. Большинство компонентов отчета являются визуальными. Многие из них мало отличаются от аналогичных компонентов страниц **Standard** (Стандартная), **Additional** (Дополнительная) и **Data Controls** (Элементы управления данными). Например, Компоненту `QRImage` соответствуют компоненты Image И `DBImage`.

11.1.1. Компонент-отчет

Главным элементом отчета является компонент-отчет `QuickRep`, представляющий собой основу, на которой размещаются другие компоненты. Компонент QuickRep обычно размещается на отдельной форме, предназначенной для создания отчета. По умолчанию он имеет имя `QuickRep1`. Если на форме размещается другой отчет (на практике обычно так не делается), он получает имя `QuickRep2` и т. д.

Компонент QuickRep при помещении его на форму имеет вид страницы формата А4, первоначально отображаемой в натуральную величину. На этапе разработки приложения можно изменить масштаб страницы и размещенных на ней компонентов с помощью свойства `Zoom` типа `integer` (значение этого свойства устанавливается в процентах, по умолчанию 100%).

Замечание

Отчет можно поместить на любую форму приложения, например, на главную. В этом случае отчет (страница) создает своего рода фон, на котором расположены управляющие элементы формы.

Компонент QuickRep связывается с набором данных Table или Query, для которого создается отчет, с помощью свойства DataSet. При этом набор данных Query может содержать записи, выбранные из разных таблиц. При печати отчета в процессе выполнения приложения набор данных должен быть открыт. Во время построения отчета можно использовать специально создаваемый набор данных и размещать его на форме, при этом источник данных DataSource не требуется. На практике компонент QuickRep часто связывается с набором данных, записи которого отображаются на форме в визуальных компонентах. В этом случае в отчет попадают записи, удовлетворяющие, например, критерию фильтрации и/или сортировки, задаваемому пользователем.

Пример. Связывание отчета с набором данных.

```
Uses Unit1;
. . .
QuickRep1.DataSet := Form1.Table1;
```

Отчет QuickRepl, находящийся на своей форме, связывается с набором данных Tablel, расположенным на форме `Form1`. В модуле формы `fmReport` должна быть задана ссылка (uses `Unit1`) на модуль формы Forml.

Отчет состоит из отдельных полос — составных частей отчета, которые определяют содержание и вид созданного документа. Полоса представляет собой элемент отчета. Каждая полоса размещается на своем месте и предназначена для отображения соответствующих компонентов отчета и вывода данных. Состав отчета и его связь с набором данных схематично показаны на рис. 11.1.

Управлять наличием полос в отчете можно с помощью свойства Bands множественного типа TQuickRepBands. При разработке приложения включение/отключение полосы выполняется установкой логического значения соответствующего подсвойства свойства Bands, например, для полосы заголовка отчета этим подсвойством является `HasTitle`. С помощью этого свойства в простой отчет можно включать следующие полосы:

- HasPageHeader — верхний колонтитул;
- HasTitie — заголовок отчета;
- HasColumnHeader — заголовки столбцов;
- HasDetail — область данных;
- HasSummary — итог отчета;
- HasPageFooter — нижний колонтитул.

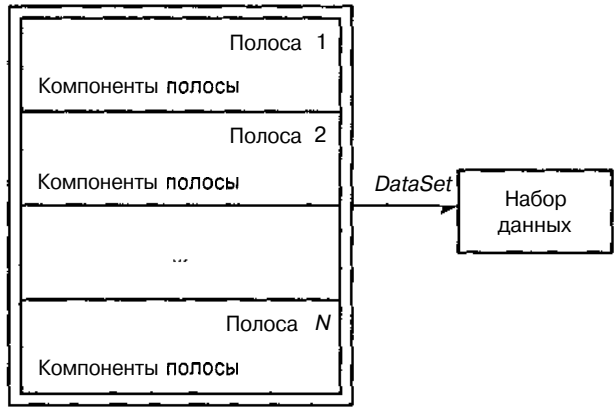


Рис. **11.1.** Состав отчета и его связь с набором данных

Перечисленные полосы можно также вставлять в отчет с помощью компонента полосы `QRBand`, при этом тип вставляемой полосы устанавливается через свойство BandType этого компонента.

Параметры страницы отчета определяет свойство Page типа TQRPage, через подсвойства которого можно настраивать такие характеристики страницы, как формат, ориентацию, размер или поля.

При необходимости разработчик может изменить параметры страницы, а также многие другие параметры отчета (например, шрифт по умолчанию) с помощью Инспектора объектов или в диалоговом окне **Report Setting** (Настройка отчета) установки параметров отчета. Оно вызывается командой **Report Setting** (Настройка отчета) контекстного меню страницы отчета или двойным щелчком на странице отчета.

Страница отчета может иметь рамку, параметры которой задает свойство Frame типа `TQRFrame`:

- `Color` — цвет (по умолчанию черный);
- width — ширина в пикселах (по умолчанию 1);
- style — стиль (по умолчанию сплошная линия);
- `DrawTop`, `DrawBottom`, DrawLeft, `DrawRight` — наличие линии сверху, снизу, слева и справа, соответственно (по умолчанию все линии отсутствуют, и рамка не рисуется).

С помощью свойства `PrinterSetting` типа `TQuickRepPrinterSettings` устанавливаются параметры *принтера:*

- FirstPage и `LastPage` — номер первой и последней печатаемой страницы;
- `Copeis` — число копий;

- Duplex — включение режима двусторонней печати;
- OutputBin — способ подачи бумаги.

Параметры принтера можно устанавливать также с помощью стандартных диалогов PrintDialog И PrinterSetupDialog ИЛИ метода PrinterSetup.

Отчет характеризуется тремя параметрами, которые задаются в свойстве Options Множественного ТИПа TQuickReportOptions:

- FirstPageHeader — печать верхнего колонтитула на первой странице отчета;
- LastPageFooter — печать нижнего колонтитула на последней странице отчета;
- Compression — отчет сохраняется в сжатом формате, при этом уменьшается объем занимаемой памяти, но снижается быстродействие.

По умолчанию СВОЙСТВО Options имеет значение [FirstPageHeader, LastPageFooter].

Свойство PrintIfEmpty типа Boolean определяет, выводить ли данные отчета для пустого набора данных. По умолчанию свойство имеет значение True, и отчет печатается, даже если в наборе данных нет ни одной записи. Это удобно, например, при печати бланков. Если свойству PrintifEmpty установить значение False, то для пустого набора данных отчет не выводится, точнее, выводится пустая страница. То есть при отсутствии записей отсутствует не только область данных, но и ряд других полос, например, заголовок отчета.

Процесс подготовки отчета к печати или просмотру может отображаться в отдельном окне. Наличие окна отображения для процесса подготовки документа определяет свойство ShowProgress типа Boolean. По умолчанию оно имеет значение True, и процесс подготовки отображается в окне на экране. Этот процесс может быть прерван нажатием клавиши <Esc>.

Для печати отчета предназначен метод Print, сразу после вызова которого отчет подготавливается к печати и направляется на установленный в системе принтер. Дополнительных подтверждений от пользователя не требуется. Метод Print может вызываться, например, при нажатии кнопки **Печать,** расположенной на форме, с которой пользователь работает.

Если компонент-отчет QuickRep связан с набором данных, записи которого выводятся в сетке DBGrid формы, то порядок записей отчета соответствует порядку записей, видимых пользователем на форме (рис. 11.2). После отбора (фильтрации) записей и/или сортировки вывод отчета происходит при нажатии кнопки **Печать,** причем учитывается новый состав и порядок следования записей.

Замечание

При формировании отчета изменяется положение указателя текущей записи, поэтому при необходимости разработчик должен предусмотреть запоминание и восстановление положения указателя, например, с помощью закладки.

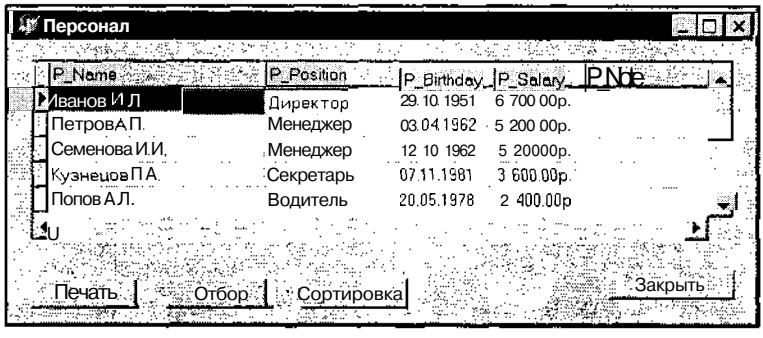


Рис. 11.2. Вид формы

Пример. Процедура печати отчета.

```
Uses uReport;
. . .
procedure TForm1.btnPrintClick(Sender: TObject);
var bm: TBookmark;
begin
// Запоминание положения указателя текущей записи
bm := Table1.GetBookmark;
// Печать отчета
fmReport.QuickRep1.Print;
// Восстановление положения указателя текущей записи
Table1.GotoBookmark(bm);
Table1.FreeBookmark(bm);
end;
```

Закладка bm используется для запоминания и восстановления положения указателя текущей записи. Модуль uReport формы fmReport отчета следует указать в списке модулей раздела uses модуля формы `Form1`, из которой выполняется печать отчета.

При печати отчета генерируются события `BeforePrint` и AfterPrint типа TQRBeforePrintEvent. С помощью обработчика первого события можно задать действия, выполняемые непосредственно перед печатью отчета, а с помощью обработчика второго события — действия, выполняемые сразу после окончания печати.

Замечание

Эти события возникают только при печати, а не при просмотре отчета. Поэтому если создан обработчик события BeforePrint, изменяющий вид и содержание отчета, то возможна ситуация, когда пользователь просматривает один вариант отчета, а напечатан он будет по-другому.

В процессе выполнения приложения для предварительного просмотра отчета перед печатью служит метод Preview, вызывающий окно просмотра (рис. 11.3). В этом окне можно:

- просмотреть отчет в различных масштабах;
- сохранить отчет в файле;
- загрузить предварительно сохраненный отчет;
- направить отчет в печать;
- выбрать принтер;
- задать параметры принтера.

Возможности метода Preview превосходят возможности метода Print, поэтому чаще выполняют именно предварительный просмотр документа, а не печать, что удобно и при отладке приложения. Печатать отчет можно непосредственно из окна предварительного просмотра.

На этапе разработки приложения также можно просмотреть отчет, выполнив команду **Preview** (Просмотр) контекстного меню отчета. Внешний вид отчета будет таким же, как при печати или в окне просмотра при выполнении приложения, за исключением отсутствия значений вычисляемых полей.

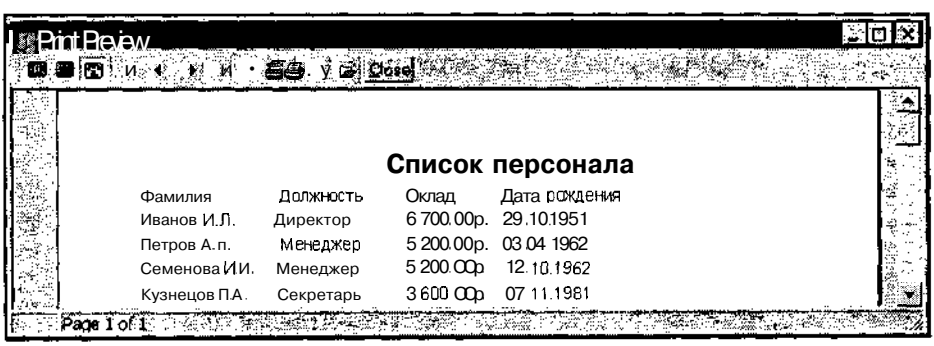


Рис. 11.3. Вид отчета в окне предварительного просмотра

При предварительном просмотре отчета в процессе выполнения приложения генерируются события OnPreview ТИПа TNotifyEvent И AfterPreview типа TQRAfterPreviewtEvent. В обработчике первого события кодируются

действия, выполняемые непосредственно перед предварительным просмотром отчета, а в обработчике второго события — действия, выполняемые сразу после окончания предварительного просмотра.
Разработчик имеет возможность создать свое окно предварительного просмотра отчета, использовав компонент QRPreview.

11.1.2. Полоса отчета

Полоса отчета (компонент QRBand) является основной составной частью (элементом) отчета, на которой размещаются другие его компоненты.
Тип полосы определяется свойством BandType типа TQRBandType, у которого можно выделить следующие значения:

- rbTitle — заголовок отчета (печатается в начале отчета под верхним колонтитулом);
- rbPageHeader — верхний колонтитул, который печатается сверху на каждой странице, в том числе на первой, если включен (по умолчанию) параметр FirstPageFooter свойства Options компонента отчета; если этот параметр выключен, то верхний колонтитул на первом листе не печатается;
- rbDetail — данные записей набора данных; выводятся для каждой записи набора данных;
- rbPageFooter — нижний колонтитул, который печатается внизу на каждой странице, в том числе на последней, если включен (по умолчанию) параметр LastPageFooter свойства Options компонента отчета; если этот параметр выключен, то нижний колонтитул на последней странице не печатается;
- rbSummary — итог отчета; выводится в конце отчета под всеми другими сведениями отчета, но выше нижнего колонтитула.

При установке типа полосы она автоматически размещается на своем месте в отчете и выравнивается по ширине страницы отчета с учетом левого и правого полей. Разработчик не имеет возможности переместить полосу на другое место с помощью мыши. Изменить ширину полосы можно косвенно, изменяя размеры страницы и полей. Высота полосы меняется путем передвижения мышью верхней или нижней рамки полосы или через установку значения свойства Height в Инспекторе объектов.

Добавить новую полосу к отчету можно следующими двумя способами:

- поместить компонент QRBand в отчет и присвоить требуемое значение свойству BandType;
- установить значение True соответствующему подсвойству свойства Bands компонента QuickRep, при этом к отчету добавляется полоса, а ее свойству BandType автоматически устанавливается нужное значение.

При создании в отчет нужно включать не более одной полосы каждого вида, так как при печати отчета "лишние" полосы одного и того же вида учитываться не будут. Например, если в отчет включены две полосы заголовка отчета (rbTitle), то ошибки компиляции не возникает, но в качестве заголовка используется первая из этих полос. Полосы определенного вида, например, полоса rbDetail, при формировании отчета создаются автоматически для каждой записи набора данных.

Каждая полоса может иметь отдельную рамку, которой управляет свойство Frame, не отличающееся от аналогичного свойства самого компонента отчета QuickRep.

При печати каждой полосы генерируются события BeforePrint типа TQRBeforePrintEvent И AfterPrint типа TQRAfterPrintEvent. Событие BeforePrint генерируется непосредственно перед печатью полосы, и его можно использовать для блокирования печати. Тип этого события описан как

```
type procedure BeforePrint (Sender: TObject; var PrintBand: Boolean) of
object;
```

Параметр PrintBand определяет, печатать ли полосу. Если вывод полосы нежелателен, то параметру следует присвоить значение False. Обычно обработчик события BeforePrint кодируется для полос данных, когда выполняется проверка данных полей текущей записи, и решение о печати полосы принимается в зависимости от выполнения определенных условий.

Замечание

В процессе выполнения приложения при вызове метода Preview **также генерируются события** BeforePrint И AfterPrint.

Пример. Процедура, в которой осуществляется управление печатью полосы.

```
procedure TfmReport.DetailBand1BeforePrint(Sender: TQRCustomBand;
var PrintBand: Boolean) ;
begin
PrintBand := Form2.Table1.FieldByName('Salary').AsFloat > 3000;
end;
```

В отчете печатаются только записи, имеющие в поле Salary значение больше 3000.

11.1.3. Компоненты, размещаемые на полосе

После создания полосы определенного типа в ней размещаются соответствующие компоненты, при этом необходимо использовать только компонен-

ты страницы **QReport** (Быстрый отчет). Размещение на полосе других компонентов, например, Label или Edit, не вызывает ошибки трансляции, но в сформированный отчет такие компоненты не попадают.

Можно разместить компонент отчета и вне полосы — непосредственно на компоненте-отчете `QuickRep`. В этом случае он будет выводиться на каждой странице отчета (например, компонент `QRImage`, содержащий логотип фирмы).

Обычно используются следующие компоненты отчета.

- `QRLabel` — надпись, содержащая текст (аналог надписи Label), которая может размещаться на любой полосе, но для полос данных обычно не используется.
- QRDBText — значение поля записи (аналог компонента DBText); обычно размещается в полосе данных.
- QRExpr — значение, формируемое на основе выражения, в котором могут использоваться значения полей записей; обычно используется для полос данных и нижних колонтитулов.
- QRSysData — системная информация, обычно используемая для итоговых полос и полос колонтитулов; ее вид определяется свойством Data типа TQRSysDataType, у которого можно выделить следующие значения:
  - `qrsDateTime` — текущие дата и время;
  - `qrsPageNumber` — номер текущей страницы;
  - qrsPageCount — общее число страниц в отчете.
- QRimage — графический образ, загружаемый аналогично графическому образу image (может быть использован в любой полосе, но обычно не размещается в полосах данных); с помощью компонента QRimage выводится, например, логотип организации; кроме того, следует отличать этот компонент и от компонента `QRDBImage`, который обычно размещается в полосе данных и отображает рисунок из поля таблицы.
- `QRShape` — геометрическая фигура, размещаемая в любой полосе.

Перед компонентом QRSysData может находиться надпись, которая указывается в свойстве Text типа `string`. Например, для значения qrsPageNumber в качестве надписи подойдет текст Страница. По умолчанию надпись отсутствует.

Компонент QRShape позволяет оформить сетку вокруг выводимых данных. Для этого в полосу помещается компонент QRShape, имеющий форму прямоугольника, а сверху него располагается текстовый элемент. Прямоугольники будут образовывать рамки сетки, для чего соседние по вертикали и горизонтали прямоугольники выравниваются так, чтобы их стороны совпадали. Если для образования сетки использовать рамку самих текстовых элементов, то необходимо учитывать, что эта рамка выводится непосредственно вокруг текста, и при стыковке соседних элементов будет отсутство-

вать межстрочный интервал. Другой вариант создания сетки — образование ее из горизонтальных линий рамки полос и вертикальных линий рамки. Однако в этом случае вертикальные линии не будут сплошными.

Выравниванием текста внутри компонентов, содержащих текст, управляет СВОЙСТВО Alignment. К таким Компонентам ОТНОСЯТСЯ, например, QRLabel и QRDBText. По умолчанию текст выравнивается по левому краю. При необходимости это значение можно изменить, установив, например, для поля даты выравнивание по центру (tacenter), а для поля цены — выравнивание ПО правому Краю (taRightJustify).

Возможностью автоматического изменения размеров содержащего текст компонента по размеру этого текста управляет свойство AutoSize. По умолчанию размер текстового компонента автоматически подстраивается под содержащийся в нем текст, так как свойство AutoSize имеет значение True. Это может привести к нарушению выравнивания столбцов отчета. Если компоненты отчета, в первую очередь элементы полосы данных, имеют рамку, то размеры этих компонентов должны быть постоянными. В таком случае свойству AutoSize устанавливается значение False.

Параметры шрифта устанавливаются через свойство Font отдельных компонентов. Для изменения шрифта сразу у всех компонентов удобно использовать свойство Font компонента QuickRep и свойство ParentFont компонентов отчета, управляющее возможностью наследовать параметры шрифта от родительского компонента. В нашем случае родительским компонентом является отчет QuickRep. По умолчанию свойство ParentFont каждого компонента отчета имеет значение True, и при изменении шрифта отчета автоматически изменяется шрифт всех его элементов.

Замечание

Если для какого-либо компонента изменяются параметры шрифта, то его свойству ParentFont присваивается значение False.

Для оформления рамки вокруг отдельного элемента отчета используется свойство Frame, управляющее наличием рамки, цветом, стилем и толщиной линий.

Многие компоненты отчета имеют событие onPrint типа TQRLabelOnPrintEvent, возникающее непосредственно перед печатью компонента. Тип этого события описан как

```
type TQRLabelOnPrintEvent = procedure (Sender: TObject;
                                       Var Value: String) of Object;
```

Параметр value обработчика события содержит значение, которое должно быть напечатано. Это значение можно изменить или отформатировать.

Действие обработчика события onPrint распространяется только на отчет и не затрагивает значения записей, находящихся в наборе данных.

Пример. Форматирование значений при печати.

```
procedure TfmReport.QRDBText3Print(Sender: TObject; var Value: String);
begin
  Value := Value + '%';
end;
```

Компонент QRDBText3 связан с числовым полем. При печати значения этого поля выводятся со знаком процента.

Обработчик события OnPrint можно использовать, например, для улучшения выравнивания компонентов при использовании вокруг них вертикальных рамок. Так, при выравнивании текста компонента QRDBText вправо (его свойство Alignment имеет значение taRightJustify) возможно наползание текста на правую линию рамки. В этом случае для улучшения читаемости текста он передвигается на несколько пробелов влево путем добавления в обработчик события строки кода:

```
Value := Value + '      ';
```

Аналогично можно поступить с текстом, выравниваемым по левой границе. При необходимости в обработчике события OnPrint выполняется и более сложное форматирование данных.

Замечание

Событие OnPrint генерируется также при вызове метода Preview, но не возникает при просмотре отчета на этапе разработки приложения.

11.2. Простой отчет

Простой отчет представляет собой отчет на основе данных из одного набора данных и содержит сведения, которые выводятся в табличном виде без дополнительных условий, например, группирования данных. Размещение и вид печатаемых в отчете данных аналогичны размещению и виду данных, отображаемых в сетке DBGrid. Отличием является то, что данные отчета размещаются не на форме, а на бумажном документе, и их нельзя редактировать.

Рассмотрим действия, выполняемые при подготовке простого отчета, на примере списка персонала (рис. 11.4). Многие из этих действий используются и при разработке отчетов других видов. Для создания простого отчета требуется:

- разместить на форме компонент QuickRep;
- создать для компонента QuickRep требуемые полосы отчета;
- разместить в полосах нужные компоненты отчета, чаще всего `QRLabel`, QRDBText И QRExpr;
- создать для какого-либо события, например, нажатия кнопки с заголовком **Печать,** обработчик, в котором вызывается метод печати или предварительного просмотра отчета.

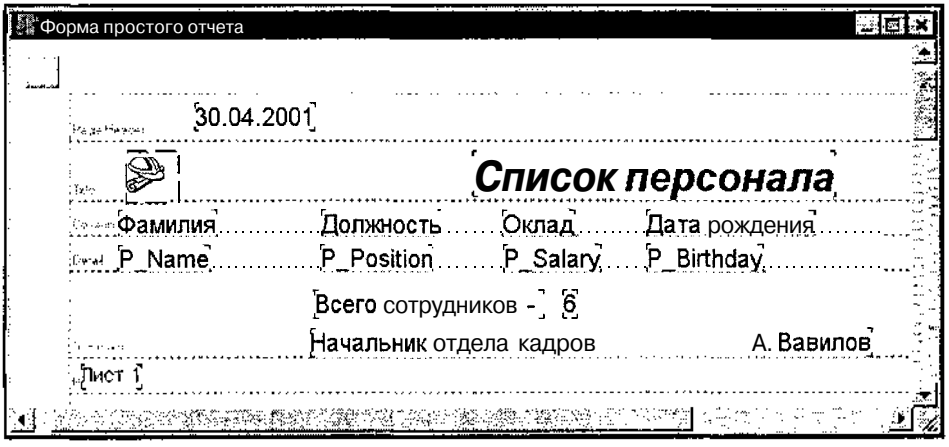


Рис. 11.4. Вид простого отчета на этапе разработки

Размещение на форме компонента отчета, установка параметров отчета, а также создание обработчика события нажатия кнопки печати рассмотрены при описании компонента QuickRep выше в этой главе.

Простой отчет может содержать следующие полосы, перечисляемые в порядке их размещения на странице:

- верхний колонтитул (`rbPageHeader`);
- заголовок отчета (`rbTitle`);
- заголовки столбцов (`rbColumnHeader`);
- данные (`rbDetail`);
- ИТОГ отчета (`rbSummary`);
- НИЖНИЙ КОЛОНТИТУЛ (`rbPageFooter`).

На этапе разработки название каждой полосы выводится серым цветом в ее левом нижнем углу.

Можно добавить к проекту шаблон простого отчета, выбрав в Хранилище объектов объект **QuickReport List** (Лист отчета), который находится на странице **Forms** (Формы).

11.2.1. Заголовок отчета

Заголовок отчета выводится один раз на первой странице сразу под верхним колонтитулом, если он есть. В полосе заголовка обычно размещаются надписи QRLabel, содержащие требуемый текст (как правило, в качестве заголовка выводится название всего отчета). При необходимости в заголовке можно разместить, например, сведения о названии, адресе и телефоне организации, а также логотип. В приведенном на рис. 11.4 примере в полосе заголовка находятся название отчета и логотип организации. Для вывода названия отчета используется компонент QRlabel, в котором набран текст Список персонала. Для названия установлен полужирный курсивный шрифт размером 20 пт. Логотип загружается в компонент QRImage. Для шрифта других компонентов рассматриваемого отчета с помощью свойства Font компонента QuickRep установлен размер 12 пт.

11.2.2. Заголовки столбцов и данные

Полосы заголовков столбцов и данных являются основными полосами, в которых размещаются компоненты, обеспечивающие табличный вывод содержимого набора данных. Заголовки столбцов выводятся на каждом листе отчета.

Для заголовков столбцов данных в полосу заголовков обычно помещаются компоненты QRiabel, в которые заносится текст, соответствующий полям данных.

Для вывода значений полей записей в полосу данных обычно помещаются компоненты QRDBText и QRExpr. Более простым является использование компонентов QRDBText, каждый из которых отображает значение связанного с ним поля. Имя набора данных указывается в свойстве DataSet, а имя поля задается в свойстве DataFileld.

На этапе разработки в отчете присутствует только одна полоса данных, но при формировании отчета отдельная полоса данных будет выведена для каждой записи отчета. Напомним, что если набор данных является пустым и не содержит записей, то область данных не выводится. Чтобы для пустого набора данных были выведены остальные полосы (кроме полос данных), свойству PrintIfEmpty компонента QuickRep устанавливается значение True (по умолчанию).

Компонент QRExpr позволяет вставлять в отчет значение выражения, рассчитываемого обычно с участием различных полей записей. Выражение заносится в свойство Expression типа string, для формирования которого удобно использовать окно **Expression Wizard** (Мастер выражений), вызываемый через Инспектор объектов.

При разработке приложения такие компоненты отчета, как QRiabel, QRDBText и QRExpr, имеют одинаковый внешний вид.

11.2.3. Итоговая полоса

Итоговая полоса отчета выводится один раз в конце отчета сразу после полосы данных. В этой полосе обычно стоят либо итоговые сведения отчета, например, средние и максимальные значения по данным какого-либо поля, либо должность и фамилия лица, утверждающего отчет. В итоговой полосе обычно размещаются компоненты: `QRlabel` — для вывода надписей и QRExpr — для вывода значений выражений.

Замечание

Если итоговая полоса не помещается на текущей странице целиком, то она отделяется от последней полосы данных и переносится на другую страницу, что не соответствует правилам делопроизводства.

В выражении для итогового компонента `QRExpr` часто используются следующие статистические функции:

- SUM — сумма значений;
- `MIN` — минимальное значение;
- `MAX` — максимальное значение;
- AVERAGE — среднее значение;
- COUNT — число записей набора данных.

Для всех функций, кроме COUNT, в скобках нужно указывать имя поля, для которого выполняется расчет. Для вывода числа записей набора данных также можно использовать компонент QRSysData со значением qrsDetailCount свойства Data.

В приведенном на рис. 11.4 примере в качестве итога выводится число записей (совместно с поясняющей надписью `Всего` сотрудников -) и начальник, утверждающий документ.

11.2.4. Колонтитулы

Колонтитулы печатаются в начале и конце каждой страницы, в них обычно выводятся сведения о дате, времени печати, а также номер страницы. Для этого в полосах колонтитулов чаще всего размещаются компоненты QRSysData, которым устанавливается требуемое значение свойства Data.

В приведенном примере отчета полоса каждого колонтитула содержит один компонент QRSysData, отображающий:

- дату — для верхнего колонтитула;
- номер страницы — для нижнего колонтитула; чтобы перед номером страницы выводился поясняющий текст, свойству Text присвоено значение `Лист`.

В полосе колонтитула можно также разместить и другие компоненты, например, `QRLabel` для вывода на каждой странице названия организации.

Предметный указатель