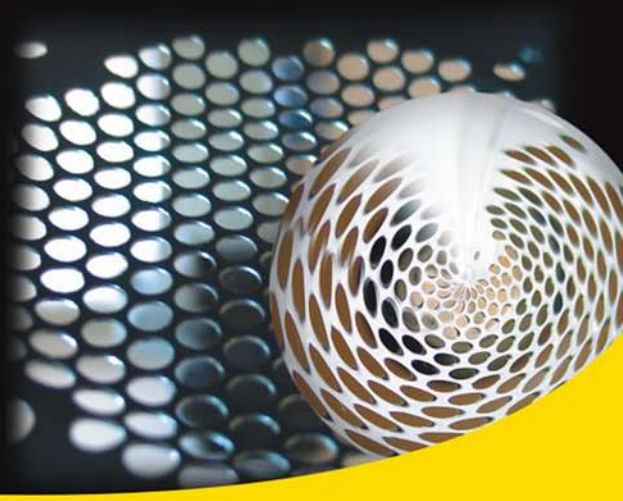
Самоучитель

Анатолий Хомоненко

Delphi .NET



Техника работы в среде Delphi 2006

Приемы создания приложений

Работа с базами данных

Примеры программ

bhv

Анатолий Хомоненко

Самоучитель Delphi .NET

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.068+800.92Delphi2006
ББК 32.973.26-018.1
Х76

Хомоненко А. Д.

Х76 Самоучитель Delphi .NET. — СПб.: БХВ-Петербург, 2006. — 464 с.: ил.

ISBN 5-94157-899-7

Рассмотрена разработка приложений в Delphi 2006 для Microsoft .NET Framework. Описаны визуальные компоненты, используемые для создания интерфейса приложений, Windows Forms и VCL.NET; компоненты и техника работы с текстовой информацией, кнопками и переключателями, а также формами и элементами меню. Уделено внимание использованию графики и мультимедиа, работе с файлами и каталогами. Приведены основные понятия, связанные с проектированием баз данных, рассмотрены элементы реляционных баз данных и техника их использования. Показаны особенности технологий ADO.NET и BDP.NET. Описано создание приложений для работы с базами данных по технологиям BDE.NET, dbGo.NET и dbExpress.NET. Показаны приемы работы с данными и подготовка отчетов с помощью генератора Rave Reports. Приводятся многочисленные примеры.

Для начинающих программистов

УДК 681.3.068+800.92Delphi2006
ББК 32.973.26-018.1

Группа подготовки издания:

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

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.07.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 37,41.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

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

ISBN 5-94157-899-7

Оглавление

Предисловие

В книге рассматривается Delphi .NET 2006, обозначающая систему программирования Delphi for the Microsoft .NET Framework, которая входит в состав продукта Borland Developer Studio 2006. Названный продукт представляет собой интегрированную среду разработки и, кроме того, включает следующие системы программирования: C# Builder, C++ Builder и Delphi for Microsoft Win32.

Delphi .NET 2006 обеспечивает объединение возможностей инструментальных средств системы Delphi версий до 7 включительно с новыми средствами и решениями, которые предоставляет платформа Microsoft .NET Framework.

С помощью системы программирования Delphi .NET 2006 можно разрабатывать приложения для платформы .NET, используя компоненты двух библиотек: Visual Component Library (VCL) для .NET — библиотеки визуальных компонентов фирмы Borland для платформы .NET; Framework Class Library (FCL) — библиотеки классов .NET Framework — разработки фирмы Microsoft. Названные библиотеки обладают во многом похожими наборами компонентов и сопоставимыми возможностями.

Книга посвящена основам работы с Delphi .NET 2006 и освоению приемов программирования с использованием визуальных средств обеих библиотек. Чтобы избежать повторов, применение каждой библиотеки дается на не перекрывающихся наборах компонентов. Рассматриваются основные средства системы, технология создания приложений для решения общих задач (от простейших приложений до не очень сложных приложений 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 .NET 2006.

Книга включает пять частей.

Часть I. Общая характеристика Delphi .NET 2006. Содержит краткое описание платформы Microsoft .NET и характеристику типов приложений Delphi .NET 2006. Описывается язык программирования Delphi .NET: типы данных, ос-

новные конструкции языка, важнейшие приемы программирования, понятия объектно-ориентированного программирования, а также ограничения и отличия языка от предыдущих версий. Рассматривается интегрированная среда Delphi .NET 2006, состав и характеристика элементов проекта приложения, компиляция и выполнение проекта, основы технологии разработки приложений.

Часть II. Приложения Windows Forms. Рассматриваются важнейшие визуальные компоненты, используемые для создания интерфейса приложений Windows Forms, разрабатываемых с помощью библиотеки Framework Class Library — библиотеки классов .NET Framework. При этом дается состав страниц **Windows Forms**, **Components** и **Dialogs** Палитры инструментов, содержащих визуальные компоненты. Все компоненты, имеющие визуальное представление, наследуются от класса `Control`. Коротко описываются общие наиболее важные, на наш взгляд, свойства, методы и события визуальных компонентов, происходящие от класса `Control`. Рассматриваются вопросы обработки событий клавиатуры и мыши, перетаскивание методом Drag and Drop.

Рассматриваются компоненты и техника работы с информацией (текстом) по ее отображению, вводу и редактированию. Описывается используемый для отображения надписей компонент `Label`, называемый меткой; компонент `LinkLabel`, задающий надпись с гиперссылкой; простейший текстовый редактор — компонент `TextBox`; редактор текста с форматированием — `RichTextBox`; обмен данными с текстовыми процессорами, а также общие свойства текстовых редакторов.

Описываются компоненты и техника работы с кнопками, списками и переключателями. В частности, рассматриваются простой и комбинированный списки и общие свойства списков, зависимый переключатель и независимый переключатель, комбинация списка и флажка. При этом освещается техника объединения, или группирования, различных элементов управления, которая может понадобиться, например, при работе с переключателями на форме или при создании панели инструментов.

Рассматривается общая характеристика форм, контейнеров и приложений. При этом характеризуются общие свойства, методы и события формы, а также важные свойства и методы контейнеров и приложения. Описываются компоненты и техника создания форм, являющихся центральной частью практически любого приложения и представляющих собой видимые окна Windows. Приводятся характеристики формы, приемы организации взаимодействия форм.

Освещаются *контейнеры*, используемые для объединения, или группирования, различных элементов управления. При этом рассматриваются так называемые *группы*, которые служат в основном для визуального выделения функционально связанных управляющих элементов, и *панели*, применяемые для группирования элементов управления, а также для создания панелей инструментов и строк состояния.

Дается общая характеристика стандартных диалогов, панели инструментов и меню. При этом указываются основные свойства, методы и события соответствую-

щих компонентов и описывается техника их создания. Эти компоненты играют важную роль в организации пользовательского интерфейса приложения.

Часть III. Приложения VCL.NET. Дается общая характеристика визуальных компонентов библиотеки VCL.NET, приводится состав страниц **Standard**, **Additional** и **Win32** Палитры инструментов. Описывается класс `TControl`, который является базовым для большинства визуальных компонентов и включает в себя общие для визуальных компонентов свойства, события и методы.

Описываются важные для отладки приложений вопросы обработки исключительных ситуаций, связанных с ошибками при выполнении программ. При этом характеризуются виды ошибок и классы исключений, рассматриваются варианты глобальной и локальной обработки исключений.

При изучении визуальных компонентов для приложений Windows Forms рассмотрены наиболее простые элементы управления. Для приложений VCL.NET описываются более сложные элементы пользовательского интерфейса: полоса прокрутки, ползунок, счетчик, строка состояния, таблица и блокноты, многостраничный и одностраничный.

Освещаются вопросы работы с графикой: рисование в процессе выполнения программ, используемые важнейшие графические компоненты `Shape`, `Bevel`, `Image`, `PaintBox` и `ImageList`. Описываются вопросы воспроизведения видеоклипов и управления мультимедийными устройствами.

Часть IV. Работа с базами данных .NET. 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.). Описывается технология создания таблиц базы данных.

Дается характеристика технологии ADO.NET, рассматривается схема доступа к данным и описываются компоненты, используемые при реализации этой технологии в Delphi .NET.

Рассматриваются провайдеры данных BDP.NET, которые при использовании технологии ADO.NET служат для соединения с базой данных, выполнения команд и получения результатов. Приводится пример приложения Windows Forms для работы с базами данных с применением технологии ADO.NET и BDP.NET.

Рассматривается подготовка отчетов (печатных документов, содержащих данные, аналогичные получаемым в результате выполнения запроса к базе данных) с помощью генератора Rave Reports. При этом описываются компоненты, предназначенные для создания отчетов; процедура печати отчета; технология подготовки простого отчета. Приводятся примеры создания отчетов.

Часть V. Работа с базами данных VCL.NET. Отмечается, что одно- и двухуровневые приложения VCL.NET могут осуществлять доступ к локальным и удаленным БД с использованием следующих технологий: BDE.NET, dbGo.NET

на основе ADO, dbExpress.NET и InterBase.NET. Описывается состав компонентов приложения для работы с базами данных VCL.NET. Рассматривается компонент-источник данных, используемый для подключения наборов данных, и визуальные компоненты, которые служат для навигации по набору данных, отображения и редактирования записей. Эти два типа компонентов применяются при создании приложений для работы с базами данных VCL.NET с использованием любой из названных технологий доступа.

Рассматриваются основные наборы данных (компоненты `Table` и `Query`), используемые для доступа к данным с помощью механизма BDE.NET. Описываются технологии выполнения операций с данными: сортировка, навигация, фильтрация, поиск записей и модификация набора данных. Приводится пример приложения для работы с базами данных по технологии BDE.NET. Поясняется работа со связанными таблицами.

Описывается технология dgGo.NET, которая в среде Delphi .NET представляет собой новое название технологии ADO, введенное с той целью, чтобы отличить технологии ADO и ADO.NET. Рассматривается установление соединения, управление соединениями и транзакциями, компоненты доступа к данным: доступ к таблицам, выполнение запросов, вызов хранимых процедур. Приводится пример приложения для работы с базами данных по технологии dgGo.NET.

Дается характеристика технологии dbExpress.NET, в основе которой лежит использование множества легковесных драйверов, компонентов, объединяющих соединения, транзакции, запросы и наборы данных, а также интерфейсов, реализующих универсальный доступ к соответствующим функциям. Рассматривается установление соединения с сервером, компоненты доступа к данным: универсальный доступ к данным, просмотр таблиц, выполнение SQL-запросов и хранимых процедур, а также отладка соединения с сервером.

Книга ориентирована на начинающих программистов, может быть полезной для специалистов-разработчиков приложений в качестве справочного издания.

ЧАСТЬ I

Общая характеристика Delphi .NET 2006

ГЛАВА 1

Состав и назначение Delphi .NET 2006

Термином Delphi .NET 2006 называют систему программирования Delphi for the Microsoft .NET Framework, которая входит в состав продукта Borland Developer Studio 2006. Названный продукт представляет собой интегрированную среду разработки и, кроме того, включает следующие системы программирования: C# Builder, C++ Builder и Delphi for Microsoft Win32. Соответствующие типы приложений могут разрабатываться либо в рамках интегрированной среды Borland Developer Studio 2006, либо с помощью перечисленных систем программирования, вызываемых с помощью отдельных модулей.

С помощью Borland Developer Studio 2006 и языка программирования Delphi можно разрабатывать приложения для двух платформ: Win32 и .NET. Мы будем вести речь о технологиях разработки приложений только для платформы .NET.

Характеристика платформы Microsoft .NET

Основу Microsoft .NET составляет .NET Framework, которая представляет собой платформу для разработки и исполнения приложений. В ее состав входят следующие основные компоненты:

- CLR (Common Language Runtime) — общеязыковая исполняемая среда;
- FCL (Framework Class Library) — библиотека классов .NET Framework.

CLR управляет выполнением кода приложений и обеспечивает службы (services), упрощающие процесс разработки приложений. В качестве примера можно назвать службы управления памятью и многоязыковой поддержки.

Библиотека классов FCL содержит множество объектно-ориентированных компонентов, служащих для разработки приложений .NET, позволяющих использовать все достоинства применения служб среды CLR.

Среда Microsoft .NET Framework обеспечивает *многоязыковую совместимость* приложений. Благодаря общеязыковой исполняемой среде приложение .NET может быть написано на любом .NET-совместимом языке программирования высокого уровня, к примеру, Visual Basic .NET и Visual C# .NET. Естественно, что теперь здесь может использоваться и язык Delphi.

При создании с помощью компилятора для платформы .NET модуля (двоичного файла) с расширением dll или exe его содержимое представляет собой *сборку* на промежуточном языке Microsoft Intermediate Language (MSIL). Кроме команд на языке MSIL, двоичный модуль .NET содержит метаданные, которые предназначены для описания всех типов в этом модуле (классов, структур, перечислений и т. д.).

В сборку может входить произвольное количество типов. Для указания определенного типа требуется обеспечить уникальность имен по отношению к другим типам в сборке. Для решения этой проблемы и предназначены пространства имен, выполняющие группирование стандартных типов и дающие возможность их использования при разработке собственных типов.

Пространство имен представляет собой логическую структуру, предназначенную для организации используемых в приложении имен с целью предотвращения конфликтов между именами из разных сборок. Например, пусть в приложении .NET есть обращения к двум разным внешним сборкам. Если при этом в каждой из них имеется тип с одним и тем же именем, то по своим функциональным возможностям эти типы различны, и смешивать их недопустимо. Следовательно, при программировании нужно указать, к какому типу и из какой сборки выполняется обращение. Для этого достаточно к имени типа добавить наименование нужного пространства имен. Например, строка `System.Int32` указывает на тип `Int32`, который принадлежит пространству имен `System`..

Типы приложений Delphi .NET 2006

С помощью системы программирования Delphi .NET 2006 можно разрабатывать приложения для платформы .NET, используя компоненты двух библиотек:

- Visual Component Library (VCL) для .NET — библиотеки визуальных компонентов фирмы Borland для платформы .NET;
- Framework Class Library (FCL) — библиотеки классов .NET Framework — "родной" для платформы .NET разработки фирмы Microsoft.

Обе указанные библиотеки обладают во многом похожими наборами компонентов и сопоставимыми возможностями. Есть заметные различия в возможностях применения названных библиотек в создании приложений для работы с базами данных и Web.

Библиотека VCL для .NET представляет собой платформу программирования для разработки приложений в среде Delphi .NET 2006 с помощью VCL. С помощью

Delphi 2006 и VCL для .NET можно разрабатывать новые приложения, а также переносить имеющиеся приложения Win32 на платформу .NET, причем с сохранением большей части кода.

Библиотека FCL.NET используется для разработки приложений для платформы .NET, называемых приложениями Windows Forms, Web-приложений с ASP.NET и Web-служб с ASP.NET.

Таким образом, в среде Delphi .NET 2006 поддерживается разработка приложений на платформе .NET с помощью языка Delphi, а также с использованием элементов управления VCL для .NET и элементов управления Windows Forms. Схематично это показано на рис. 1.1. Как видим из приведенной схемы, в Delphi 2006 поддерживается создание приложений для работы с базами данных для всех известных нам по предыдущим версиям системы технологий доступа, но применительно к платформе .NET (ADO.NET, dbExpress.NET, BDE.NET, IBX.NET, DataSnap.NET). В случае использования ADO.NET применяются провайдеры данных фирмы Borland — BDP.NET.

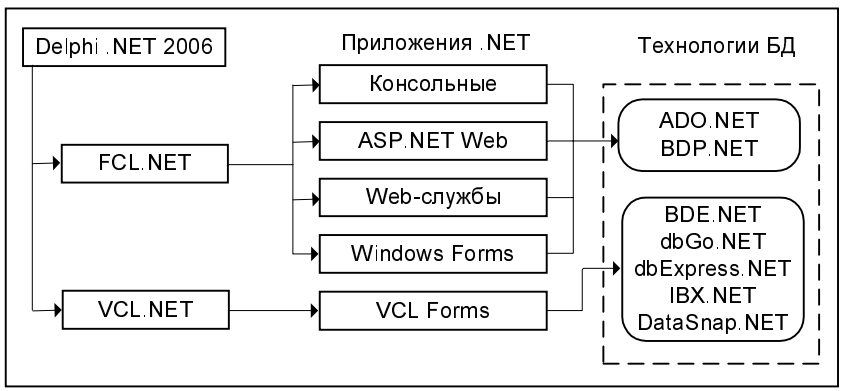


Рис. 1.1. Варианты приложений Delphi .NET 2006

Библиотека VCL для .NET представляет собой большое подмножество наиболее общих классов в составе библиотеки VCL для Win32.

Платформа .NET разработана так, чтобы обеспечить согласование с любым языком, входящим в ее состав. Поэтому код Delphi, оперирующий с классами и функциями VCL для Win32, во многих случаях может быть перекомпилирован на платформу .NET с минимальными изменениями.

Классы библиотеки VCL для .NET принадлежат пространству имен `Borland.VCL`. При этом классы, связанные с базами данных, размещены в пространстве имен `Borland.VCL.DB`. Библиотека классов времени выполнения (Runtime Library Classes) находится в пространстве имен `Borland.VCL.RTL`. Файлы модулей (unit) были собраны в соответствующие пространства имен `Borland.VCL`. Некоторые

модули были перемещены, тем не менее, пространства имен позволяют облегчить поиск объектов, требуемых для достижения нужной функциональности приложения. Файлы с исходным кодом для всех объектов Delphi 2006 размещены в подкаталоге C:\Program Files\Borland\BDS\4.0\source.

Замечание

В книге мы рассматриваем разработку приложений только для платформы .NET. Поэтому, говоря о разработке приложений с помощью библиотеки VCL.NET, для краткости мы будем считать синонимами фразы "Приложение VCL", "Приложение VCL.NET" и "Приложение VCL Forms".

ГЛАВА 2

Язык программирования Delphi .NET

Основу языка программирования Delphi составляет язык Object Pascal. При этом система программирования Delphi накладывает ряд ограничений. Язык программирования Delphi .NET подразумевает использование платформы .NET, а это приводит к ряду дополнительных ограничений и отличий. В данной главе мы рассмотрим основные средства и приемы программирования на языке Delphi .NET, оговаривая специально эти ограничения и отличия.

Основные понятия

Алфавит

Алфавит языка Delphi включает в себя следующие символы:

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

  Комбинации специальных символов образуют следующие *составные* символы:

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

Словарь языка

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

Слова подразделяются на:

- ключевые слова;
- директивы;
- стандартные идентификаторы;
- идентификаторы пользователя.

Ключевые (зарезервированные) слова являются составной частью языка, имеют фиксированное написание и однозначно определенный смысл, изменить который программист не может. Полный перечень ключевых слов включает следующие слова:

and	else	inherited	packed	then
array	end	initialization	procedure	threadvar
as	except	inline	program	to
asm	exports	interface	property	try
begin	file	is	raise	type
case	final	label	record	unit
class	finalization	library	repeat	unsafe
const	finally	mod	resourcestring	until
constructor	for	nil	sealed	uses
destructor	function	not	set	var
dispinterface	goto	object	shl	while
div	if	of	shr	with
do	implementation	or	static	xor
downto	in	out	string	

В редакторе кода ключевые слова выделяются полужирным шрифтом.

Слова `above`, `private`, `protected`, `public`, `published` и `automated` являются зарезервированными словами внутри объявления класса, а в остальных случаях воспринимаются как директивы. Слова `at` и `on` имеют специальное значение и должны обрабатываться как зарезервированные слова.

Директивы имеют контекстную зависимость значения от своего местоположения в исходном коде программы. К числу директив относятся следующие слова:

absolute	dynamic	local	platform	requires
abstract	export	message	private	resident
assembler	external	name	protected	safecall
automated	far	near	public	stdcall
cdecl	forward	nodefault	published	stored
contains	implements	overload	read	varargs
default	index	override	readonly	virtual
deprecated	inline	package	register	write
dispid	library	pascal	reintroduce	writeonly

Стандартные идентификаторы служат для обозначения следующих заранее определенных разработчиками конструкций языка:

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

В отличие от ключевых слов, директиву и любой из стандартных идентификаторов можно переопределить, но поскольку это может привести к ошибкам, то их лучше использовать без изменений. Примерами стандартных идентификаторов являются слова `Sin`, `Pi`, `Real`.

Идентификаторы задаются программистом для обозначения имен меток, констант, переменных, процедур, функций и типов данных. Они должны удовлетворять следующим требованиям:

- составляются из букв и цифр;
- идентификатор начинается с буквы, исключением являются метки, которыми могут быть целые числа без знака в диапазоне 0 .. 9999;
- в идентификаторе допускаются строчные и прописные буквы, компилятор интерпретирует их одинаково; использование в идентификаторах специальных символов не допускается, поэтому для наглядности отдельные составляющие идентификатора полезно выделять прописными буквами, например, `NumberLines` или `btnOpen`;
- между двумя идентификаторами в программе должен быть, по крайней мере, один разделитель.

Структура программы

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

Заголовок находится в начале программы и имеет вид:

```
Program <Имя программы>;
```

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

В общем случае описательная часть состоит из следующих разделов:

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

В конце каждого из указанных разделов ставится точка с запятой.

Замечание

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

Структуру программы в общем случае можно представить 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 программе любой из разделов описания и объявления может отсутствовать. Разделы описаний и объявлений, кроме раздела подключения модулей, который

располагается сразу после заголовка программы, могут встречаться в программе произвольное число раз и следовать в произвольном порядке. При этом все описания и объявления элементов программы должны быть сделаны до того, как они будут использованы. Рассмотрим подробнее отдельные разделы программы.

Раздел *подключения модулей* включает зарезервированное слово `Uses` и список имен подключаемых стандартных и пользовательских библиотечных модулей. Формат этого раздела:

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

Например,

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

Раздел *объявления меток* включает зарезервированное слово `Label` и список имен меток. Формат данного раздела:

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

Например,

```
Label met1, met2, 11, 67;
```

Раздел *объявления констант* начинается ключевым словом `Const`, за которым следует ряд операторов, присваивающих константам значения. Имя константы отделено от выражения знаком равенства, в конце оператора ставится точка с запятой. Формат этого раздела:

```
Const <идентификатор1> = <Выражение>;
           ...
          <идентификаторN> = <Выражение>;
```

Например,

```
Const st1 = 'Hello'; ch = '3'; pi1 = 3.1415;
```

Тип константы распознается 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. В конце описания ставится точка с запятой. Формат раздела:

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

Например,

```
Type char2 = ('a'..'z');
     massiv = array[1..50] of real;
     week = 1..7;
```

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

Каждая переменная программы должна быть объявлена. Объявление обязательно предшествует использованию переменной. Раздел *объявления переменных* начинается с ключевого слова `Var`, после которого через запятые перечисляются имена переменных и через двоеточие указывается их тип. Формат раздела:

```
Var <идентификаторы> : <тип>;
       ...
    <идентификаторы> : <тип>;
```

Например,

```
Var  a, u7:   real;
     simv:    char;
     i1,j2:   integer;
```

При компиляции операторов объявления переменных под них выделяется соответствующий их типу участок памяти для размещения без присваивания начальных значений.

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

Раздел *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.

Комментарии

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

Примеры комментариев:

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

Допускается вложение комментариев, ограниченных различными символами, например,

```
(* { Вложенные комментарии } *)
```

При компиляции комментарий игнорируется и не оказывает влияния на выполнение программы.

Типы данных

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

Тип определяет множество значений, которые могут принимать элементы данных, и совокупность допустимых над ними операций. Например, значения `34` и `67` относятся к целому типу, их, соответственно, можно умножать, складывать, делить и выполнять другие арифметические операции, а значения `'abcd'` и `'sdfh123'` относятся к строковому типу, и их можно 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 — это тоже группы, включающие 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.

Большое 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, т. к. целое число автоматически преобразовывается в вещественное (но не наоборот).

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, а к структурированным — оператор разветвления и оператор цикла.

Операторы разделяются между собой точкой с запятой. Наличие между операторами нескольких точек с запятой не является ошибкой, т. к. они обозначают пустые операторы. Лишняя точка с запятой в разделе описаний и объявлений будет уже синтаксической ошибкой.

Точка с запятой может не ставиться после слова `begin` и перед словом `end`, т. к. они рассматриваются как операторные скобки, а не как операторы. В условных операторах и операторах выбора точка с запятой не ставится после слова `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.

Директивы компилятора

Текст программы может содержать специальные команды, называемые *директивами компилятора*, которые служат для управления режимами компиляции. Ди-

ративы компилятора заключаются в фигурные скобки, а в их начале ставится символ `$`. С помощью директив компилятора можно, например, задать способы интерпретации строковых типов, размер стека или подключить файл ресурса.

Программист обычно управляет режимами компиляции через окно параметров проекта, включая или выключая соответствующие переключатели на страницах **Compiler**, **Compiler Messages**, **Linker** и **Directories/Conditionals**. При этом установленные значения параметров сохраняются в файле параметров проекта (DOF).

Простые типы данных

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

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

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

Для значений и имен порядковых типов определены следующие функции:

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

Кроме того, к ним применимы следующие процедуры:

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

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

Некоторые простые типы делятся на фундаментальные (физические) и общие. *Фундаментальные типы* закладываются в язык и не зависят от особенностей конкретного компьютера. *Общие типы* соответствуют одному из конкретных фундаментальных типов, и их использование считается более предпочтительным, т. к. при этом компилятор создает более эффективный код.

Целочисленные типы

Целочисленные типы предназначены для целых чисел и могут быть фундаментальными и общими. Фундаментальные целочисленные типы Delphi приводятся в табл. 2.1.

Таблица 2.1. *Фундаментальные целочисленные типы*

Обозначение	Диапазон	Представление в памяти
`Shortint`	–128 .. 127	1 байт, со знаком
`Smallint`	–32 768 .. 32 767	2 байта, со знаком
`Longint`	–2 147 483 648 .. 2 147 483 647	4 байта, со знаком
`Int64`	–263.. 262	8 байтов, со знаком
`Byte`	0 .. 255	1 байт, без знака
`Word`	0 .. 65 535	2 байта, без знака
`Longword`	0 .. 4 294 967 295	4 байта, без знака

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

Таблица 2.2. *Общие целочисленные типы*

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

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

Литерные типы

Значениями *литерного типа* являются элементы из набора литер, т. е. отдельные символы. Для символов также имеются фундаментальные и общие типы. Фундаментальные литерные типы представлены типами `AnsiChar` и `WideChar`.

Символ типа `AnsiChar` занимает один байт, а для кодирования символов используется код ANSI Американского национального института стандартов (American National Standards Institute). Символ типа `WideChar` занимает два байта, а для кодирования символов используется международный набор символов Unicode. На-

бор символов Unicode включает в свой состав более 60 тысяч элементов и, кроме прочего, позволяет кодировать символы национальных алфавитов. Первые 256 символов Unicode совпадают с кодом ANSI.

В языке Delphi определен один общий тип `Char`, который эквивалентен типу `AnsiChar`.

Для операций с символами имеются следующие функции:

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

Логические типы

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

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

Перечислимые типы

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

Формат описания перечислимого типа:

```
Type <имя типа> = (<Имя1>, ..., <имяN>);
```

Например,

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

Тип `Day` описан явно и для него определены значения — дни недели. Переменные `d1`, `d2` могут принимать одно из перечисленных значений. Попытка присвоить им любое другое значение вызовет программную ошибку. Второй тип определен без указания имени путем перечисления значений при объявлении переменной `Season`. Последняя является переменной перечислимого типа и может принимать только одно из 4-х указанных в скобках значений.

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

Интервальные типы

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

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

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

Например:

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

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

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

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

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

Вещественные типы

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

Таблица 2.3. *Фундаментальные вещественные типы*

Обозначение	Минимальное значение	Максимальное значение	Точность (число цифр мантиссы)	Память, байт
`Real48`	2.9×10^{-39}	1.7×10^{38}	11—12	6
`Single`	1.7×10^{-45}	3.4×10^{38}	7—8	4
`Double`	5.0×10^{-324}	1.7×10^{308}	15—16	8
`Extended`	3.6×10^{-4951}	1.1×10^{4932}	19—20	10
`Comp`	$-2\times10^{63}+1$	$2\times10^{63}-1$	19—20	8
`Currency`	–922337203685477.5808	922337203685477.5807	19—20	8

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

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

Типы `Comp` и `Currency` служат для представления вещественных чисел с фиксированной точкой и введены для точных расчетов денежных сумм. Тип `Comp` фактически представляет целые числа, но относится к вещественным типам. При присваивании переменной этого типа вещественного значения оно автоматически округляется до ближайшего целого.

К выражениям вещественных типов применимы следующие функции:

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

Структурные типы данных

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

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

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

Строки

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

Таблица 2.4. *Физические строковые типы*

Обозначение	Максимальная длина символов	Память
`ShortString`	255	2 байта .. 256 байт
`AnsiString`	примерно 2×10^{31}	4 байта .. 2 Гбайта
`WideString`	примерно 2×10^{30}	4 байта .. 2 Гбайта

Тип `ShortString` представляет собой строку, которая фактически является массивом из 256 элементов — `array [0 .. 255]`. Нулевой байт этого массива (строки) указывает длину строки. В ранних версиях языка подобная строка обозначалась типом `String`, тип `ShortString` введен в Object Pascal для обеспечения совместимости с этими версиями.

Язык Delphi поддерживает также подтипы типа `ShortString`, максимальная длина которых изменяется в диапазоне от 0 до 255 символов. Обозначение их выполняется с помощью целого в квадратных скобках, указываемого справа от ключевого слова `string`. Например, операторы:

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

или

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

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

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

Отметим, что операционная система Windows наряду с однобайтовым набором символов обеспечивает поддержку многобайтовых наборов символов, таких как Unicode. При однобайтовом наборе символов каждый байт представляет один символ. В многобайтовом наборе некоторые символы представляются одним байтом, другие символы представляются более чем одним байтом. А именно, первые 128 символов многобайтового набора символов соответствуют карте

7-битовых ASCII-символов, и любой из байтов, порядковое значение которого больше чем 127, является ведущим байтом многобайтового символа.

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

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

Общим типом является тип `String`, который может соответствовать как типу `ShortString`, так и типу `AnsiString`, что определяется директивой компилятора `$H`. По умолчанию используется `{$H+}`, и тип `String` равен типу `AnsiString`. В файле параметров проекта знак "+" директивы (а также других директив) соответствует записи `H=1`, а знак "-" — записи `H=0`. Разработчик обычно управляет интерпретацией типа `String` через окно параметров проекта, включая или отключая флажок **Huge strings** (Большие строки). Допустимо это только в приложениях для Win32.

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

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

Массивы

Массивом называется упорядоченная индексированная совокупность однотипных элементов, имеющих общее имя. Элементами массива могут быть данные различных типов, включая структурированные. Каждый элемент массива однозначно определяется *именем* массива и *индексом* (номером этого элемента в массиве) или индексами, если массив многомерный. Для обращения к отдельному элементу массива указывается 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 размерность массива (одномерный, двумерный и т. д.), при этом размерность не ограничивается. В математике аналогом одномерного массива является вектор, а двумерного — матрица. Индексы элементов массива должны принадлежать порядковому типу. Разные индексы одного и того же массива могут иметь различные типы. Наиболее часто типом индекса массива является целочисленный.

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

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

Например,

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

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

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

```
Array of <Тип элементов>;
```

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

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

Приведем пример использования динамического массива.

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

Здесь после описания динамического массива, состоящего из вещественных чисел, определяется его размер, равный 100. Каждому элементу присваивается зна-

чение, равное номеру этого элемента в массиве. Так как нумерация элементов массива начинается с нуля, то номер последнего из них равен не 100, а 99. После завершения цикла размер массива увеличивается до двухсот.

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

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

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

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

Множества

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

```
Set of <Тип элементов>;
```

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

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

Таблица 2.5. *Операции над множествами*

Операция	Наименование	Тип результата	Результат
+	Объединение множеств	set of	Неповторяющиеся элементы первого и второго множеств
-	Разность множеств	set of	Элементы первого множества, не принадлежащие второму
*	Пересечение множеств	set of	Элементы, общие для обоих множеств
=	Эквивалентность	boolean	True, если множества эквивалентны
<>	Неэквивалентность	boolean	True, если множества не эквивалентны

***Таблица 2.5** (окончание)*

Операция	Наименование	Тип результата	Результат
<=	Проверка вхождения	boolean	True, если первое множество входит во второе
>=	Проверка включения	boolean	True, если первое множество включает второе

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

Пример использования множеств:

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

Здесь определяется множественный тип MonhtDays и объявляются две переменные Color и Day множественного типа, над которыми выполняются операции объединения и вычитания. В четвертом операторе присваивания к множеству Color повторно добавляется значение Black. Такое присваивание корректно, однако значение множества при этом не изменяется. В шестом операторе присваивания из пустого множества Day вычитается элемент, в результате выполнения этого оператора значение множества также не изменяется.

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.

Записи

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

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

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

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

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

Пример использования записи:

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

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

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

```
type Tpoint = record
  X: Longint;
  Y: Longint;
end;

Trect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: Tpoint);
end;
```

Здесь тип `Tpoint` является фиксированной записью, а тип `Trect` — вариантной записью (рис. 2.1).

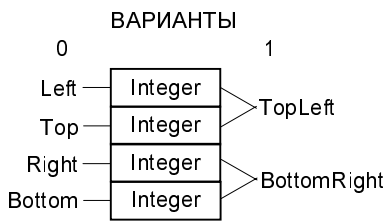


Рис. 2.1. Тип `Trect` — вариантная запись

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

Файлы

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

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

В зависимости от типа элементов различают текстовые, типизированные и нетипизированные файлы. *Текстовый* файл содержит строки символов переменной длины, *типизированный* файл составляют элементы указанного типа (кроме файлового), а в *нетипизированном* файле находятся элементы, тип которых не указан. Описание файловой переменной, предназначенной для работы с файлом, должно соответствовать типу элементов файла.

Например:

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

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

Замечание

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

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

Другие типы данных

Указатели

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

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

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

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

Пример объявления переменных-указателей:

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

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

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

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

Например:

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

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

В модулях `System` и `SysUtils` имеется большое количество типов указателей, например, `PansiString`, `Pstring`, `Pcurrency`, `Pinteger` или `Pvariant`, которые можно использовать без их предварительного описания. Модуль `System` вызывается компилятором автоматически, а имя модуля `SysUtils` автоматически вносится средой Delphi в список подключаемых модулей раздела `Uses` для каждой формы.

Небезопасный код

С позиций платформы .NET указатели считаются небезопасными, так как реализуют прямой доступ к памяти. В связи с этим при использовании указателей следует сообщить компилятору об использовании небезопасного кода. Для этого в содержащий небезопасный код модуль нужно поместить директиву `{$UNSAFECODE ON}`, а подпрограммы с небезопасным кодом следует описывать с помощью ключевого слова `unsafe`.

Например,

```
procedure unsafecodeProc; unsafe;
begin
// код с указателями
. . .
end;
```

Небезопасный код может быть полезен при переходе от платформы Win32 на платформу .NET. При этом появляется возможность последовательного перевода кода по частям, оставляя исходный код небезопасным до тех пор, пока все приложение не будет переведено на безопасный код.

Процедурные типы

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

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

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

```
type TnotifyEvent = procedure (Sender: Tobject) of object;
```

Процедурный тип используется, например, для назначения обработчиков событий, в том числе и через Инспектор объектов.

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

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

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

Вариантные типы

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

Для описания переменной вариантного типа используется ключевое слово `Variant`. Такой переменной можно присваивать значения целочисленных (кроме

Int64), вещественных, символьных, строковых и логических типов. Все эти типы считаются совместимыми с типом Variant, и в случае необходимости операция преобразования типа выполняется компилятором автоматически.

Пример использования вариантного типа:

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

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

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

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

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

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

Первый из операторов в блоке begin выполняет создание вариантного массива (длины 3) вариантных элементов и назначает его вариантной переменной X. Второй параметр в вызове функции VarArrayCreate определяет базовый тип элементов массива (в примере вариантный). При вызове стандартной функции

`VarArrayOf` создается вариантный массив (из четырех элементов) с набором целочисленных значений, который с помощью оператора присваивания назначается третьему элементу вариантного массива `X`.

Замечание

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

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

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

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

Выражения

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

- операндов;
- знаков операций;
- круглых скобок.

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

Операнды представляют собой данные, над которыми выполняются действия. В качестве операндов могут использоваться константы (литералы), переменные, элементы массивов и обращения к функциям.

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

Круглые скобки используются для изменения порядка выполнения операций.

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

Арифметические выражения

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

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

Таблица 2.6. *Бинарные арифметические операции*

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

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

Кроме функции `Sin()`, имеется также много других тригонометрических функций, в том числе и обратных, например, `ArcSin()`.

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

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

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

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

Поясним использование арифметических операций для целочисленных операндов на примере.

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

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

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

- `Shl` — сдвиг влево;
- `Shr` — сдвиг вправо;
- `And` — И (побитовое умножение);
- `Or` — ИЛИ (побитовое сложение);

- `Xor` — побитовое исключающее ИЛИ;
- `Not` — НЕ (побитовое отрицание).

Особенностью побитовых операций является то, что они выполняются над операндами поразрядно (табл. 2.7).

Таблица 2.7. *Побитовые операции*

Операция	Описание	Бит 1	Бит 2	Бит результата
`Not`	Отрицание	0		1
		1		0
`And`	Умножение	0	0	0
		0	1	0
		1	0	1
		1	1	1
`Or`	Сложение	0	0	0
		0	1	1
		1	0	1
		1	1	1
`Xor`	Исключающее ИЛИ	0	0	0
		0	1	1
		1	0	1
		1	1	0

Рассмотрим пример использования побитовых операций.

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

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

При этом, например, результат выполнения операции `And` получается следующим образом:

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

Замечание

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

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

Отметим также, что для целых типов имеется функция `Odd(X): Boolean`, которая анализирует четность выражения `X`.

Логические выражения

Результатом *логического выражения* является логическое значение `True` или `False`. Логические выражения чаще всего используются в 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,
- `<>` — не равно.

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

Замечание

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

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

Таблица 2.8. *Логические операции*

Операция	Описание	Операнд 1	Операнд 2	Результат
`not`	Отрицание	`False`		`True`
		`True`		`False`
`and`	Логическое И	`False`	`False`	`False`
		`False`	`True`	`False`
		`True`	`False`	`False`
		`True`	`True`	`True`
`or`	Логическое ИЛИ	`False`	`False`	`False`
		`False`	`True`	`True`
		`True`	`False`	`True`
		`True`	`True`	`True`
`xor`	Исключающее ИЛИ	`False`	`False`	`False`
		`False`	`True`	`True`
		`True`	`False`	`True`
		`True`	`True`	`False`

Примеры логических выражений:

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

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

Строковые выражения

Результатом строкового выражения является строка символов. Для строк можно применять операцию "`+`", выполняющую соединение (конкатенацию) двух строк.

Имеются также следующие функции:

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

Процедуры:

- `Insert(Source: String; var S: String; Index: Integer)` — вставка строки `Source` в строку `S`, начиная с позиции `Index`;
- `Delete(var S: String; Index, Count: Integer)` — удаление из строки `S` подстроки символов длиной `Count`, начиная с позиции `Index`;
- `Val(S; var V; var Code: Integer)` — преобразование строки `S` в число `V`, тип которого зависит от представления числа в строке; параметр `Code` возвращает код результата операции (0, если операция выполнена успешно);
- `Str(X [: Width [: Decimals ]]; var S)` — преобразование значения численного выражения `X` в строку `S`.

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

- `IntToStr (Value: Integer): String` — преобразование значения целочисленного выражения `Value` в строку;
- `StrToInt (const S: String): Integer` — преобразование строки `S` в целое число;
- `FloatToStr(Value: Extended): String` — преобразование значения вещественного выражения `Value` в строку;
- `StrToFloat(const S: string): Extended` — преобразование строки `S` в вещественное число;
- `DateToStr(Date: TDateTime): String` — преобразование значения даты в выражении `Date` в строку;
- `TimeToStr(Time: TDateTime): String` — преобразование значения времени в выражении `Time` в строку;
- `StrToDateTime(const S: String): TDateTime` — преобразование строки `S` в дату и время;
- `StrToDate(const S: String): TDateTime` — преобразование строки `S` в дату;

- `StrToTime(const S: String): TDateTime` — преобразование строки `S` во время;
- `UpperCase(const S: String): String` — перевод символов строки `S` в верхний регистр;
- `LowerCase(const S: String): String` — перевод символов строки `S` в нижний регистр;
- `Trim(const S: String): String` — удаление 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 функции `Now()`, `Date()` и `Time()`, возвращающие текущие значения даты и времени.

Примеры строковых выражений:

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

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

Простые операторы

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

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

Оператор присваивания

Оператор присваивания является основным оператором языка. Он предписывает вычислить выражение, заданное в его правой части, и присвоить результат переменной, имя которой стоит в левой части оператора. Переменная и выражение должны иметь совместимый тип, например, вещественный и целочисленный (но не наоборот). Допустимо присваивание значений данных любого типа, 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 значение выражения сначала вычисляется, а потом присваивается указанной переменной. Поэтому при условии, что x является числовой переменной и имеет определенное значение, допустимой и правильной будет следующая конструкция:

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

Примеры операторов 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;
```

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 оператор. При этом для осуществления перехода оператор, на который передается управление, должен быть помечен меткой. Метка, стоящая перед оператором, отделяется от него двоеточием.

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

```
goto <Метка>;
```

Пример использования оператора перехода:

```
Label m1;
...
goto m1;
...
m1:  <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 структуриро-

ванных операторов, а также операторам, находящимся в других блоках (процедурах, функциях).

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.

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 данный оператор будет рассмотрен при изучении процедур.

Структурированные операторы

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

- составной оператор;
- условный оператор;
- оператор выбора;
- операторы цикла;
- оператор доступа.

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 составного оператора:

```
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, при этом никаких ограничений на глубину вложенности составных операторов не накладывается.

Условный оператор

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

```
if <Условие> then <Оператор1> [else <Оператор2> ];
```

Условие представляет собой выражение логического типа. Оператор работает следующим образом: если условие истинно (имеет значение `True`), то выполняется `оператор1`, в противном случае — `оператор2`. Оба оператора могут быть составными.

Допускается запись условного оператора в сокращенной форме, когда слово `else` и оператор после него отсутствуют. В этом случае при невыполнении условия управление сразу передается на оператор, следующий за условным.

Для организации ветвления на три и более направлений 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 условных операторов:

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

Оператор выбора

Оператор выбора является обобщением условного оператора и позволяет сделать выбор из произвольного числа имеющихся вариантов, т. е. организовать ветвление на произвольное число направлений. Этот оператор состоит из выра-

жения, 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:

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

Пример оператора 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`.

Операторы цикла

Операторы цикла служат для организации циклов (повторов). *Цикл* представляет собой последовательность операторов, которая может выполняться более одного раза. Группу повторяемых операторов называют телом цикла.

В Delphi имеются следующие виды операторов цикла:

- с параметром;
- с предусловием;
- с постусловием,
- перебора членов контейнера.

Обычно, если количество повторов известно заранее, то применяется оператор цикла с параметром, в противном случае — операторы с пост- или предусловием. Последний вид оператора цикла применяется в случае необходимости перебора членов контейнера (массива, множества, строки или коллекции).

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 с соответствующим шагом всех значений параметра цикла от начального до конечного. Шаг параметра всегда равен `1` для первого формата цикла и `-1` — для второго, т. е. значение параметра последовательно увеличивается (`for ... to`) или уменьшается (`for ... downto`) на единицу при каждом повторении цикла.

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

Примеры циклов с параметром:

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

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

Оператор цикла с постусловием

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

```
repeat
<Оператор1>;
 ...
<ОператорN>;
until <Условие>;
```

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

В теле цикла может находиться произвольное число операторов без операторных скобок `begin` и `end`. По крайней мере один из операторов тела цикла должен влиять на значение условия, в противном случае произойдет зацикливание.

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

```
var x:   integer;
    sum: real;
    m:   array[1..10] of real;
...
x:=1; sum:=0;
```

```
repeat
  sum := sum + m[x];
  x := x + 1;
  until(x < 10);
```

Оператор цикла с предусловием

Оператор цикла с предусловием целесообразно использовать в случаях, когда число повторений тела цикла заранее неизвестно и тело цикла может ни разу не выполняться. Этот оператор аналогичен оператору `repeat ... until` с той лишь разницей, что проверка условия стоит 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`, т. е., в отличие от цикла с постусловием, цикл выполняется при значении логического выражения `True`.

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

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

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

Оператор цикла перебора членов контейнера

Оператор цикла перебора членов контейнера имеет следующий формат:

```
for <Элемент> in <Контейнер> do <Оператор>;
```

С помощью приведенной конструкции удобно выполнять перебор всех членов некоторого контейнера, например, массива, множества, строки или коллекции. При этом счетчик повторений тела цикла задается автоматически. Тип членов (`<Элемент>`) должен соответствовать типу контейнера (`<Контейнер>`).

Например, копирование строк типа `String` и `ShortString` с помощью рассматриваемой конструкции можно выполнить следующим образом:

```
var
Ch: Char;
St1, St2: String;
Cw: WideChar;
Sw1, Sw2: WideString;
begin
St1 := 'Anything string';
St2 := '';
for Ch in St1 do
St2 := St2 + Ch;
Sw1 := 'Anything wide string';
Sw2 := '';
for Cw in Sw1 do
Sw2 := Sw2 + Cw;
end.
```

Как видим из приведенного примера, для строк типа `String` элементы должны иметь тип `Char`, а для строк типа `WideString` элементы должны иметь тип `WideChar`.

Рассматриваемую конструкцию цикла поддерживают следующие классы и их наследники: `TList`, `TCollection`, `TStrings`, `TInterfaceList`, `TComponent`, `TMenuItem`, `TCustomActionList`, `TFields`, `TListItems`, `TTreeNodes` и `TToolBar`.

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`.

Например:

```
// 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 красной линией толщиной в пять пикселов рисуется прямоугольник. Для обращения к свойствам и методу (процедуре) поверхности рисования формы удобно использовать оператор доступа (второй вариант).

Для наглядности обычно будем писать составные имена полностью (как в первом варианте приведенного примера).

В операторе доступа допускается указывать несколько имен объектов.

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

Такой формат эквивалентен следующей конструкции:

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

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

Подпрограммы

Подпрограмма представляет собой группу 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 делится на два этапа:

- описание подпрограммы;
- вызов подпрограммы.

Любая подпрограмма должна быть предварительно описана, после чего допускается ее вызов. При описании подпрограммы определяется ее имя, список параметров и выполняемые подпрограммой действия. При вызове 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 собственные 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 в качестве результата своей работы может возвращать под своим именем некоторое значение и, соответственно, может поэтому 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 применяется последний способ. При этом различают параметры и аргументы. *Параметры* (формальные параметры) являются элементами подпрограммы и используются при описании операций, выполняемых подпрограммой.

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

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

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

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 состоит из имени процедуры и списка аргументов, заключенного в круглые скобки.

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

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

// Описание пользовательской процедуры ChangeStr
procedure ChangeStr(var Source: string; const char1, char2: char);
label 10;
var n: integer;
begin
10:
n:=pos(char1, Source);
if n > 0 then begin
   Source[n]:=char2;
   goto 10;
   end;
end;

var str1:              string;
    Year, Month, Day:  word;
begin
// Вызов процедуры DecodeDate
DecodeDate(Now, Year, Month, Day);
str1:=Edit1.Text;
// Вызов пользовательской процедуры ChangeStr
ChangeStr(str1, '1', '*');
Edit1.Text:=str1;
end;
```

Процедура `DecodeDate`, служащая для разложения даты на отдельные составляющие (год, месяц и день), может быть использована без предварительного описания, т. к. она уже описана в модуле `SysUtils`. Процедура `ChangeStr` выполняет замену в строке `Source` всех вхождений символа, который задает параметр `char1`, на символ, задаваемый параметром `char2`.

Предварительное описание пользовательской процедуры `ChangeStr` выполнено непосредственно в обработчике события нажатия кнопки `Button1`. Это описание следует вынести за пределы обработчика, в таком случае процедуру `ChangeStr` можно будет вызывать не только из данного обработчика.

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

Функции

Описание *функции* состоит из заголовка и блока. *Заголовок* 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 параметры) ] : <Тип результата>;
```

Возвращаемое значение может иметь любой тип, кроме файлового.

Блок функции представляет собой локальный блок, по структуре аналогичный блоку процедуры. В теле функции 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 допускается указывать переменную `Result`, которая создается в качестве синонима для имени функции. В отличие от имени функции, переменную `Result` следует использовать в выражениях блока функции. С помощью этой переменной можно в любой момент получить внутри блока доступ к текущему значению функции.

Замечание

Имя функции в принципе также можно использовать в выражениях блока функции, однако это приводит к рекурсивному вызову 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 в выражения в качестве операнда.

Для примера 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 примера предыдущего раздела. Вызов функции используется в операторе присваивания.

Рекурсивные подпрограммы

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

Рассмотрим, например, вычисление факториала числа:

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

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

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

Параметры и аргументы

Параметры являются элементами подпрограммы и используются при описании производимых в ней действий. 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 вызове подпрограммы были подставлены вместо формальных параметров.

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 `d` и `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 фор-

мального параметра отражаются на аргументе. Таким образом можно вернуть результаты из подпрограммы по окончании ее работы.

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

```
function F2 (var e1; const t2): integer;
```

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

Модули

Кроме программ, структуру которых мы только что рассмотрели, средства языка позволяют создавать модули. В отличие от программы, *модуль* не может быть автономно запущен на выполнение и содержит элементы, например, переменные и подпрограммы, которые допускается использовать в программе или в других модулях. Для того чтобы можно было использовать средства модуля, его необходимо подключить, указав имя этого модуля в разделе `uses`. Типичными примерами модулей являются `System` и `SysUtils`, содержащие большое количество стандартных подпрограмм (некоторые из них уже были рассмотрены). Напомним, что для каждой формы приложения создается отдельный модуль.

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

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

Модуль имеет следующую структуру:

```
Unit <Имя модуля>;

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

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

// Раздел инициализации
Initialization
<Операторы>

// Раздел деинициализации
Finalization
<Операторы>
End.
```

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

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

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

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

Особенности объектно-ориентированного программирования

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

Основные концепции ООП

К основным концепциям ООП относятся следующие:

- инкапсуляция;
- наследование;
- полиморфизм.

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

Наследование — это процесс порождения новых объектов-потомков от существующих объектов-родителей, при этом потомок берет от родителя все его поля, свойства и методы. В дальнейшем наследуемые поля, свойства и методы можно использовать в неизменном виде или переопределять (модифицировать).

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

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

Пример объявления нового класса:

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

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

Классы и объекты

В языке Object Pascal классы — это специальные типы данных, используемые для описания объектов. Соответственно *объект*, имеющий тип какого-либо класса, является *экземпляром* 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 описания класса:

```
type
   TColorCircle = class(TCircle);
      FLeft,
```

```
      FTop,
      FRight,
      FBottom: Integer;
      Color: TColor;
  end;
```

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

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

```
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, доступные 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, видимые в любом месте программы, где доступен сам класс.

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

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

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

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

Пример обращения к полям объекта:

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

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

Поля

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

Пример описания полей:

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

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

При создании новых классов класс-потомок наследует все поля родителя, при этом удаление или переопределение этих полей невозможно, но допускается добавление новых. Таким образом, чем дальше по иерархии какой-либо класс находится от родительского класса, тем больше полей он имеет.

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

Свойства

Свойства 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 описания свойств:

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

Для доступа к полям `Fcode`, `FSign` и `FNote`, которые описаны в защищенном разделе и недоступны другим классам, используются свойства `Code`, `Sign` и `Note` соответственно.

Методы

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

Например, описание метода `Button1Click` будет выглядеть так:

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

Метод, объявленный в классе, может вызываться различными способами, что зависит от вида этого метода. Вид метода определяется модификатором, который

указывается в описании класса после заголовка метода и отделяется от заголовка точкой с запятой. Приведем некоторые модификаторы:

- `virtual` — виртуальный метод;
- `dynamic` — динамический метод;
- `override` — переопределяемый метод;
- `message` — обработка сообщения;
- `abstract` — абстрактный метод.

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

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

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

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

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

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

Приведем примеры на использование конструктора и деструктора:

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    procedure PenChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ...
  end;
// Описание конструктора Create класса TShape
constructor TShape.Create(Owner: TComponent);
```

```
begin
  inherited Create(Owner);  // Инициализация унаследованных частей
  Width := 65;              // Изменение унаследованных свойств
  Height := 65;
  FPen := TPen.Create;      // Инициализация новых полей
  FPen.OnChange := PenChanged;
end;
```

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

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

Сообщения и события

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

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

Поля этой записи содержат следующую информацию:

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

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

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

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

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

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

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

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

```
type
  TForm1 = class(TForm)
  // Объявление метода обработки сообщения
  procedure MyPaint(Var Param); message WM_Size;
  end;
...
// Код метода обработки сообщения
procedure TForm1.MyPaint(Var Param);
```

```
begin
// Вызов метода-предка
inherited;
// Очистка поверхности формы
Form1.Refresh;
// Вывод красной рамки
Form1.Canvas.Pen.Color:=clRed;
Form1.Canvas.Brush.Style:=bsClear;
Form1.Canvas.Rectangle(0, 0, Form1.ClientWidth, Form1.ClientHeight);
end;
```

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

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

Пример назначения 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 следующим образом:

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

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

Так как события являются свойствами, то их значения можно изменять в процессе выполнения приложения. То есть можно *динамически* изменять реакцию объекта на одно и то же событие. При этом допускается назначать обработчик собы-

тия одного объекта другому объекту или его событию, если совпадают типы событий. Подобная возможность обеспечивается с помощью указателя на класс.

Кроме явно задаваемых параметров, например параметра `Sender`, методу всегда передается указатель на вызвавший его экземпляр класса. Этим указателем является параметр `Self`.

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

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

Рассмотрим следующий пример:

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

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

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

Динамическая информация о типе

Объекты содержат динамическую информацию о собственном типе (RunTime Type Information, RTTI) и наследовании, которая доступна во время выполнения программы и которую можно использовать, например, для проверки принадлежности объекта к тому или иному типу. Поскольку для каждого объекта допустимы только определенные операции, зависящие от его типа, то такая проверка позволяет предотвратить опасные ситуации, связанные с выполнением недопустимых действий.

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

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

```
<Объект> is <Класс>
```

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

Оператор `as` предназначен для приведения одного типа к другому и используется в выражении вида:

```
<Объект> as <Класс>
```

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

Рассмотрим следующий пример на неявное приведение типа:

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

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

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

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

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

Явное приведение типа выполняется с помощью следующей конструкции:

```
<Тип>(<Объект>)
```

Пример явного приведения типа:

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

Название компонента, на котором пользователем было выполнено нажатие, изменяет свое название на `Кнопка`. Для выполнения присваивания тип компонента приводится к типу `TButton`.

Библиотека визуальных компонентов

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

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

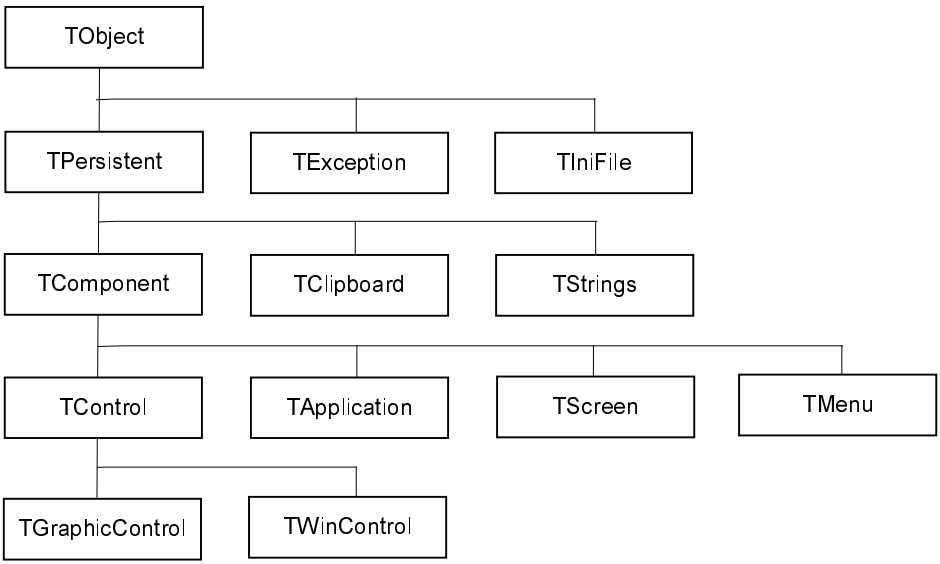


Рис. 2.2. Фрагмент иерархии классов

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

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

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

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

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

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

- свойства:
  - `Components` — список принадлежащих компонентов;
  - `ComponentCount` — число принадлежащих компонентов;
  - `ComponentIndex` — номер компонента в списке принадлежащих компонентов;
  - `ComponentState` — состояние текущего компонента;
  - `Name` — имя компонента;
  - `Owner` — владелец компонента;
  - `Tag` — целое значение, хранимое вместе с компонентом;
- методы:
  - `DestroyComponents` — разрушить все принадлежащие компоненты;
  - `Destroying` — уведомить принадлежащий компонент о его разрушении;
  - `FindComponent` — найти компонент в списке `Components`.

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

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

Таковы наиболее общие классы библиотеки визуальных компонентов.

ГЛАВА 3

Интегрированная среда Delphi .NET 2006

Средства интегрированной среды

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

- Главное окно (**Project№ — Borland Developer Studio 2006 — WinForm№**).
- Страница приглашения (**Welcome Page**).
- Палитра инструментов (**Tool Palette**).
- Конструктор формы (**Form Designer**).
- Редактор кода (**Code Editor**).
- Инспектор объектов (**Object Inspector**).
- Менеджер проектов (**Project Manager**).
- Хранилище, или репозиторий, объектов (**Object Repository**).
- Проводник данных (**Data Explorer**).
- Окна отладки (**Debug Windows**).
- Списки для выполнения (**To-Do Lists**).
- Окно структуры проекта (**Structure**).

Вид интегрированной среды разработки (пользовательский интерфейс) может различаться в зависимости от настроек. После загрузки системы и создания при-

ложения Windows Forms интерфейс Delphi .NET 2006 выглядит так, как показано на рис. 3.1.

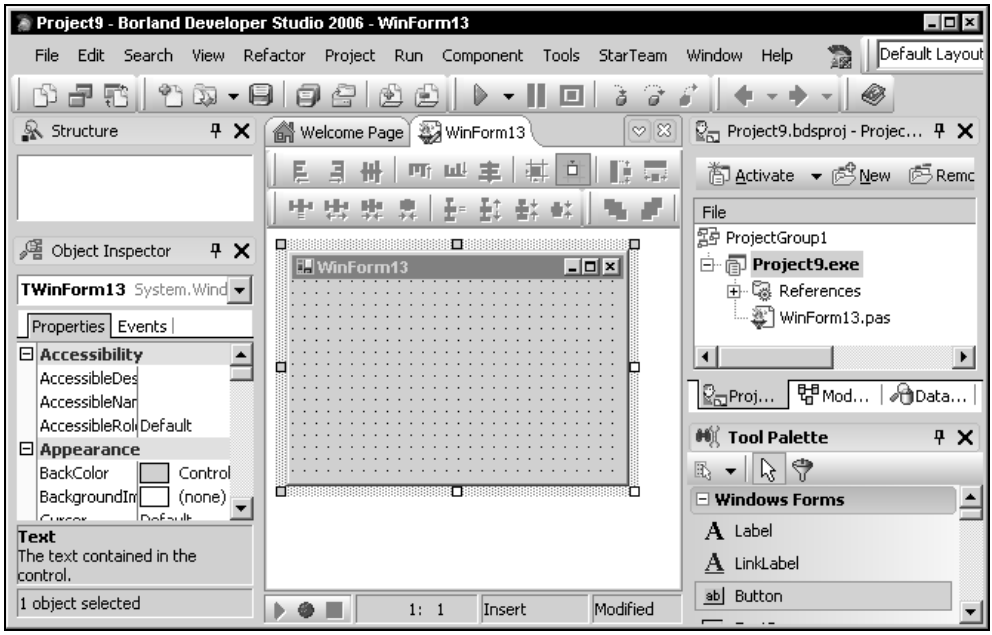


Рис. 3.1. Вид интерфейса интегрированной среды

На экране, кроме окон указанных средств разработки, могут присутствовать и другие окна, отображаемые при вызове соответствующих средств. Окна Delphi можно перемещать, изменять их размеры и убирать с экрана (кроме главного окна), а также состыковывать между собой. Дадим краткую характеристику основным средствам разработки интегрированной среды.

Главное окно

Delphi является однодокументной средой и позволяет одновременно работать только с одним приложением (проектом приложения). Название проекта приложения выводится в строке заголовка *главного окна* (рис. 3.2).

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

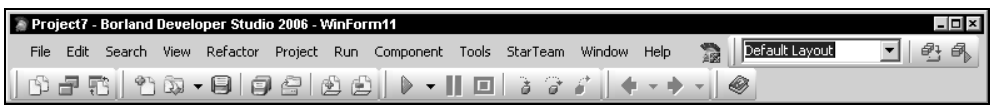


Рис. 3.2. Главное окно

Главное окно Delphi включает:

- главное меню;
- панели инструментов (**Toolbars**).

Главное меню содержит обширный набор команд для доступа к функциям Delphi, основные из которых рассматриваются далее.

Вызвать многие команды главного меню можно также с помощью комбинаций клавиш, указываемых справа от названия соответствующей команды. Например, команду **View** | **Window List** (Просмотр | Список окон) можно вызвать с помощью комбинации клавиш <Alt>+<0>.

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

Всего имеется 13 панелей инструментов:

- **Standard** (Стандартная);
- **Debug** (Отладка);
- **Desktop** (Рабочий стол);
- **Custom** (Пользователь);
- **Align** (Выравнивание);
- **Spacing** (Размещение);
- **Position** (Позиция);
- **Personality** (Информация о продукте);
- **Browser** (Обозреватель);
- **HTML Design** (Конструирование HTML);
- **HTML Format** (Формат HTML);
- **HTML Table** (Таблица HTML);
- **View** (Просмотр).

В среде можно управлять отображением панелей инструментов и настройкой 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.

Более широкие возможности по настройке панелей инструментов и главного меню предоставляет показанное на рис. 3.3 диалоговое окно **Customize** (Настройка), вызываемое одноименной командой контекстного меню панелей инструментов. С его помощью можно скрыть или отобразить требуемую панель инструментов, изменить состав кнопок на ней, а также выбрать режим отображения всплывающих подсказок и комбинаций клавиш для кнопок.

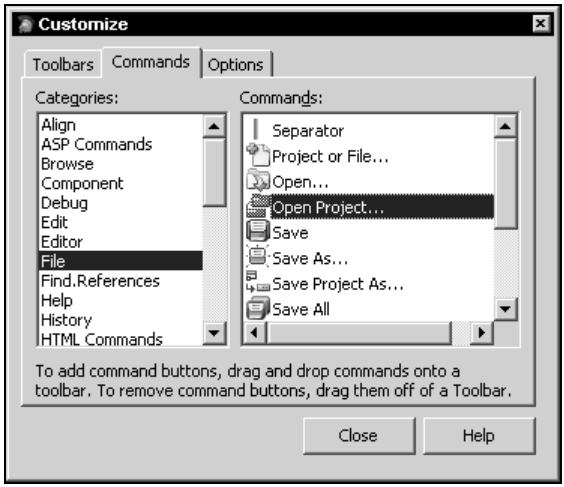


Рис. 3.3. Диалоговое окно настройки панелей инструментов

Страница приглашения

Страница приглашения содержит ряд параметров и ссылок для выбора. В частности, на ней отображается список недавно выполненных проектов. С ее помощью можно также получить доступ к справочной информации в Интернете, открыть Хранилище данных для выбора нужного шаблона при создании приложения, вызвать диалоговое окно открытия проекта или группы проектов.

Палитра инструментов

Палитра инструментов после загрузки среды первоначально находится в правой нижней части экрана (см. рис. 3.1). Ее содержимое зависит от выбора страницы приглашения (**Welcome Page**) или поверхности Конструктора (**Designer Surface**) созданного приложения.

При выборе *страницы приглашения* Палитра инструментов содержит множество вариантов приложений, документов или сценариев, которые можно создать с помощью Delphi .NET 2006, например, **IntraWeb Test Application for .NET** (рис. 3.4). Варианты создаваемых приложений разбиты по типам. Если для создаваемого приложения активизировать вкладку **Code** (Код) или **History** (История), то на Палитре инструментов также будет содержаться множество вариантов приложений, документов или сценариев.

При выборе *поверхности Конструктора* созданного приложения (вкладки **Design** Конструктора приложения) Палитра инструментов содержит множество *компонентов*, размещаемых в создаваемых формах (см. рис. 3.1). *Компоненты* являются "строительными блоками", из которых конструируются формы прило-

жения. Все компоненты разбиты на категории (categories), каждая из которых в Палитре инструментов располагается на отдельной странице, а сами компоненты представлены значками. Нужная страница компонентов выбирается из списка после щелчка мыши на кнопке **Categories** (Категории).

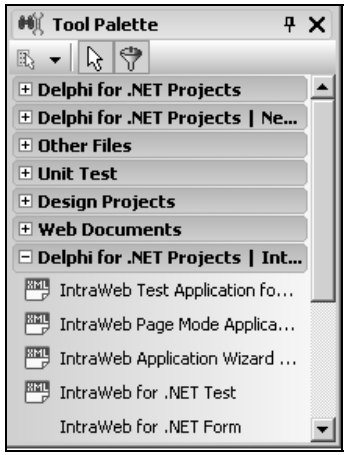


Рис. 3.4. Варианты приложений на Палитре инструментов

Текущий состав категорий компонентов в Палитре инструментов зависит от вида конструируемого приложения, документа или сценария. К примеру, при создании HTML-страницы (HTML Page) доступна всего одна категория компонентов — **HTML Elements**.

В случае создания приложения Windows Forms в составе Палитры инструментов доступен следующий набор категорий:

- **Windows Forms** — формы Windows;
- **Data Components** — работы с информацией из баз данных;
- **Borland Data Provider** — провайдер данных фирмы Borland;
- **DB Web** — элементы управления DB Web;
- **Enterprise Code Objects** — создание приложений ECO;
- **Components** — компоненты;
- **Dialogs** — создание стандартных диалоговых окон;
- **Data Controls** — создание элементов управления данными;
- **Crystal Report** — создание отчетов Crystal Report.

Более подробно основные компоненты будем рассматривать по ходу изложения.

Состав компонентов можно *настраивать* с помощью диалогового окна **Installed .NET Components** (Установленные компоненты .NET). Это окно (рис. 3.5) вызы-

вается с помощью одноименной команды контекстного меню Палитры инструментов или одноименной команды пункта **Component** (Компонент) главного меню. С помощью вкладки **.NET Components** (Компоненты .NET) окна (рис. 3.5) можно выполнить подключение/отключение компонентов нужной категории.

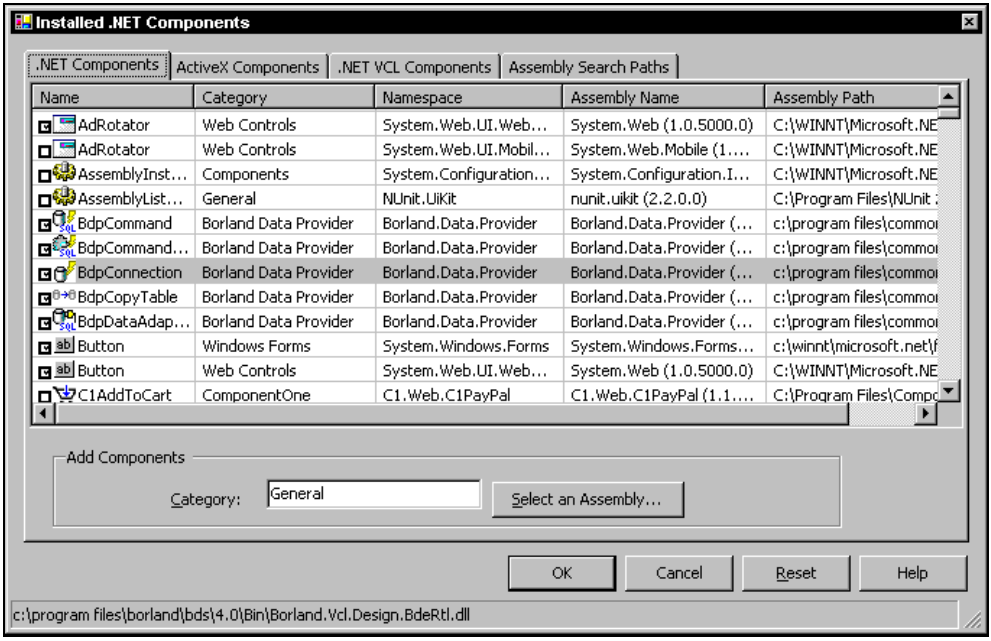


Рис. 3.5. Вкладка **.NET Components** в окне настройки состава компонентов

Управление подключением/отключением выбранного компонента выполняется с помощью флажка в поле **Name** (Имя) на указанной вкладке (рис. 3.5). Так, при установке флажка происходит добавление соответствующего компонента на страницу с категорией, указанной в поле **Category**.

С помощью вкладок **ActiveX Components** (Компоненты ActiveX) и **.NET VCL Components** (Компоненты .NET VCL) диалогового окна **Installed .NET Components** (рис. 3.5) аналогичным образом можно выполнить подключение/отключение компонентов категории **ActiveX** и компонентов **.NET VCL** соответственно.

Конструктор формы

Окно (поверхность) Конструктора формы первоначально находится в центре экрана и имеет заголовок **<имя формы>** (для приложения Windows Forms) (см. рис. 3.1) или **<имя модуля>** (для приложения VCL.NET). В нем выполняется проектирование формы, по ходу которого в форму из Палитры помещаются необходимые компоненты. Проектирование заключается в визуальном конструиро-

вании формы. Конструктор формы при этом остается как бы "за кадром", и разработчик имеет дело непосредственно с формой.

Вызов окна Конструктора формы выполняется выбором вкладки **Design** (Конструирование). Для приложений Windows Forms окно Конструктора формы (рис. 3.6) включает две составляющие:

- окно формы или просто "форму", на которой размещаются *визуальные* компоненты, например, `Button` или `Label`. Оно расположено в верхней части поверхности Конструктора;
- индикатор компонентов (находится под окном формы), на которой размещаются *не визуальные* компоненты формы, например, `Timer` или `BdpConnection`.

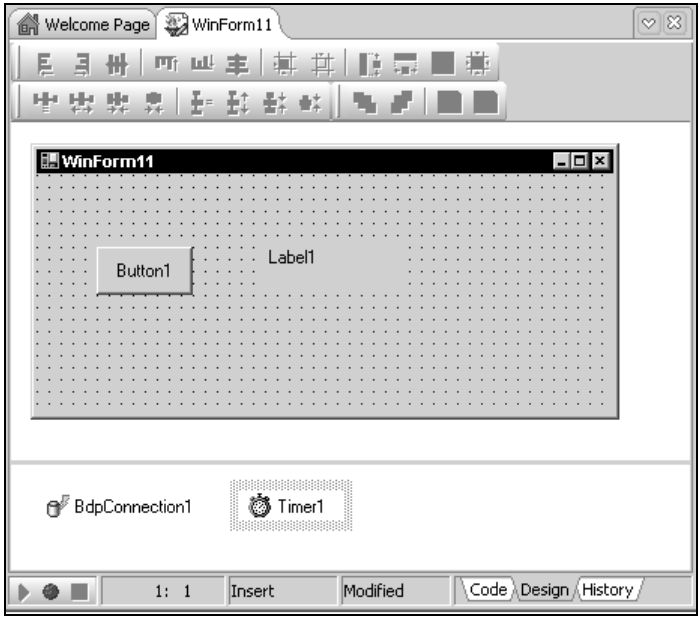


Рис. 3.6. Вид поверхности Конструктора формы

При разработке приложений VCL.NET как *визуальные*, так и *не визуальные* компоненты размещаются на форме.

Редактор кода

Окно *Редактора кода* **<имя формы>** (для приложения Windows Forms) или **<имя модуля>** (для приложения VCL.NET) находится на вкладке **Code** (Код) окна Конструктора формы (рис. 3.7). Редактор кода представляет собой текстовый редактор, с помощью которого можно редактировать текст модуля и другие текстовые файлы приложения, например файл проекта. Редактор кода поддер-

живает также просмотр и редактирование других элементов приложения, имеющих расширения имен pas, htm, aspx и txt. Каждый редактируемый файл находится в окне Редактора кода на отдельной странице, доступ к которой осуществляется щелчком на соответствующем значке. Переключаться между окнами Формы и Редактора кода удобно с помощью клавиши <F12>.

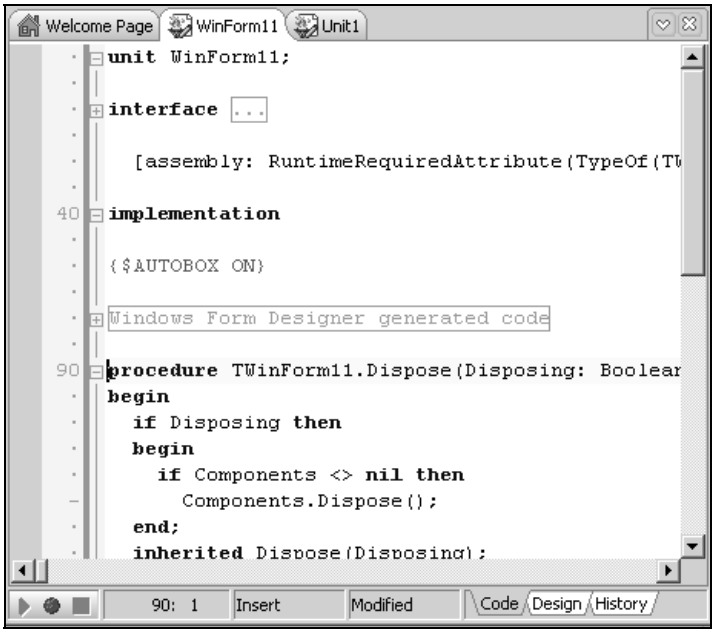


Рис. 3.7. Редактор кода

Код в окне Редактора для удобства разбит на блоки (`unit`, `implementation` и т. д.), каждый из которых может быть развернут или свернут с помощью значка + и – соответственно.

Инспектор объектов

Окно *Инспектора объектов* после запуска интегрированной среды находится под главным окном Delphi в левой части экрана. Оно отображает свойства и события объектов для текущей формы, например `WinForm11`. Окно можно открыть командой **View** | **Object Inspector** (Просмотр | Инспектор объектов) или нажатием клавиши <F11>. Окно Инспектора объектов имеет две вкладки: **Properties** (Свойства) и **Events** (События).

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

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

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

Инспектор объектов позволяет группировать свойства и события по категориям или в алфавитном порядке. Свойства (и их значения) отображаются различными цветами. В Инспекторе объектов содержатся и свойства, предназначенные только для чтения.

Отображение названий свойств и событий по категориям (рис. 3.8) выполняется командой **Arrange | by Category** (Расположить | По категориям) контекстного меню Инспектора объектов. Командой **Arrange | by Name** (Расположить | По имени) задается расположение по алфавиту.

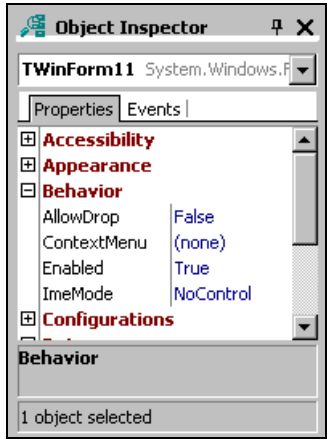


Рис. 3.8. Расположение свойств и событий по категориям

По умолчанию Инспектор объектов отображает все свойства и события объектов. Можно развернуть или свернуть отображение свойств некоторой категории с помощью значка + и – соответственно в левом поле строки с этой категорией (например, **Behavior**) в окне Инспектора объектов.

Менеджер проектов

Менеджер проектов служит для управления проектами и составными частями разрабатываемого приложения. Вызов Менеджера проектов выполняется командой **View** | **Project Manager** или комбинацией клавиш <Ctrl>+<Alt>+<F11>. Вид окна Менеджера проектов показан на рис. 3.9.

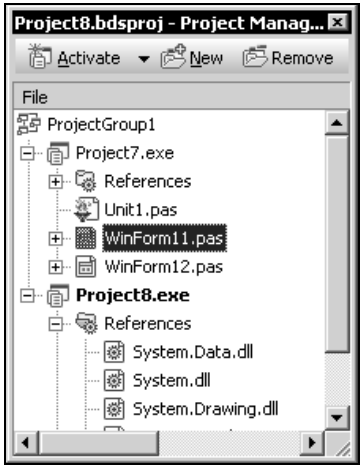


Рис. 3.9. Окно Менеджера проектов

Как и ранее, система Delphi является SDI-приложением (Single Document Interface, однодокументный интерфейс) и в каждый момент времени способна обрабатывать только один проект. Менеджер проектов частично устраняет это ограничение и позволяет работать с группой, которая объединяет несколько проектов. В группу удобно объединять проекты, например, при одновременной разработке приложений клиента и сервера или при разработке динамической библиотеки и вызывающего ее приложения. Файл группы проектов имеет расширение bdsgroup и имя ProjectGroup1 по умолчанию.

С помощью Менеджера проектов можно выполнить следующие действия:

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

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

В составе группы один проект является *активным*, т. е. команды компиляции и запуска приложения применяются именно к нему. Активизировать проект можно, выбрав его в раскрывающемся списке под заголовком окна Менеджера проекта, или командой **Activate** (Активизировать) контекстного меню этого проекта,

или нажатием одноименной кнопки. Имя активного проекта выделяется полужирным шрифтом.

Хранилище объектов

Система программирования позволяет многократно использовать одни и те же объекты в качестве шаблонов для разработки других приложений. Для хранения таких объектов применяется Хранилище объектов. В нем находятся различные объекты, например, шаблоны консольных приложений, форм, HTML-страниц.

Вставить в приложение новый объект можно, открыв командой **File** | **New** | **Other** (Файл | Новый | Другой) окно **New Items** (Новые элементы) для выбора нового объекта в хранилище (рис. 3.10). Это окно можно также открыть нажатием кнопки **New** панели инструментов Менеджера проектов.

Все объекты в хранилище (см. рис. 3.10) размещены по категориям, например: **Delphi for .NET Projects** — шаблоны приложений .NET и их компонентов. Для добавления нового объекта к текущему проекту необходимо выбрать нужную категорию и указать объект. В примере на рис. 3.10 выбран объект **Windows Forms Application** (Приложение Windows Forms), принадлежащий категории **Delphi for .NET Projects** (Проекты Delphi for .NET).

Отметим, что объекты формы Windows, элемента управления пользователя и класса можно добавить к проекту также через подменю **File** | **New**, в котором содержатся команды добавления к проекту названных объектов.

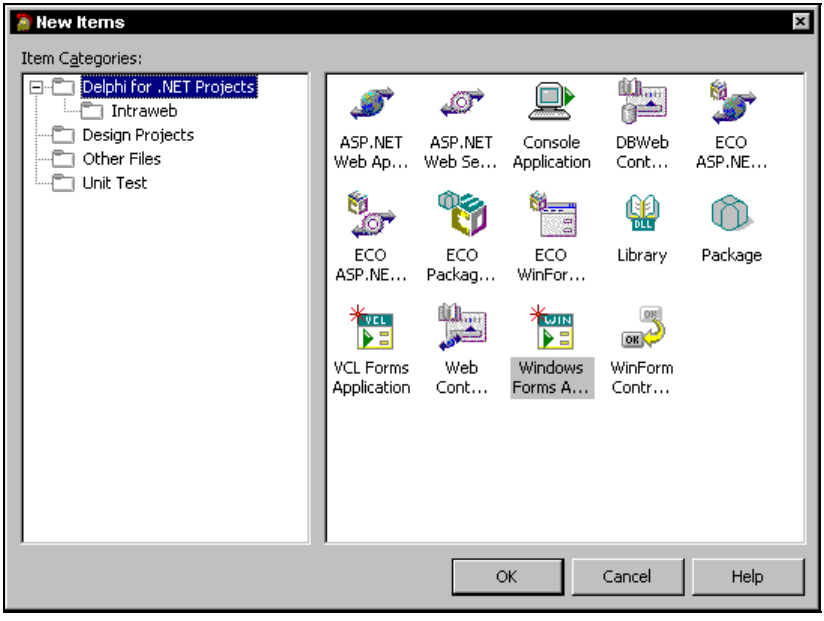


Рис. 3.10. Окно выбора объекта в Хранилище

Проводник данных

Проводник данных (**Data Explorer**) позволяет для различных серверов баз данных (Interbase, Oracle, DB2 и MSSQL) выполнять просмотр различных объектов баз данных (таблиц, хранимых процедур, триггеров, индексов). Кроме того, с помощью контекстного меню Проводника данных можно выполнять операции с соединениями баз данных (обновление, добавление, переименование и удаление). Задать отображение Проводника данных можно путем выбора вкладки **Data Explorer** (Проводник данных) в правой верхней части экрана (см. рис. 3.1).

Список для выполнения

Список для выполнения (**To-Do List**) служит для установки перечня задач, которые должны быть выполнены для текущего проекта. Задать отображение названного списка можно по команде **View** | **To-Do List** (Вид | Список для выполнения).

Окно структуры проекта

Окно структуры проекта (**Structure**) (рис. 3.11) служит для отображения информации о структуре исходного кода проекта приложения. Окно находится в левой верхней части экрана, вызывается по команде **View** | **Structure** (Вид | Структура). Отображение информации в этом окне осуществляется при условии активизации Редактора кода путем выбора вкладки **Code** (Код) или с помощью клавиши <F12>.

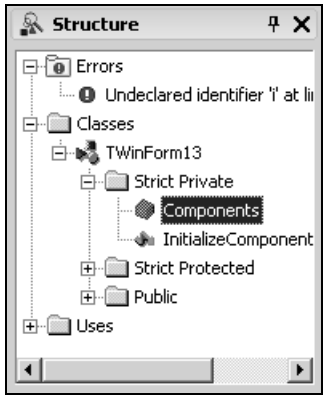


Рис. 3.11. Окно структуры проекта

С помощью окна структуры проекта легко получить представление о строении исходного кода или кода HTML, отображаемого в окне Редактора кода. При выборе некоторого объекта в окне структуры в Редакторе кода отображается соответствующий этому объекту фрагмент кода. Кроме того, с помощью этого окна удобно контролировать синтаксические ошибки в исходном коде. На рис. 3.11

показан пример сообщения о синтаксической ошибке (необъявленный идентификатор), отображаемого в окне структуры проекта.

Параметры инструментальных средств

Настройка параметров инструментальных средств интегрированной среды выполняется в диалоговом окне **Options** (Параметры), открываемом командой **Tools** | **Options** (Сервис | Параметры). В левой части окна выбирается инструментальное средство и категория настраиваемых параметров, при этом в правой части окна отображаются соответствующие сделанному выбору параметры для настройки. Для примера на рис. 3.12 показан вид окна **Options** применительно к случаю выбора для настройки параметров цвета (**Color**) Редактора кода (**Editor Options**).

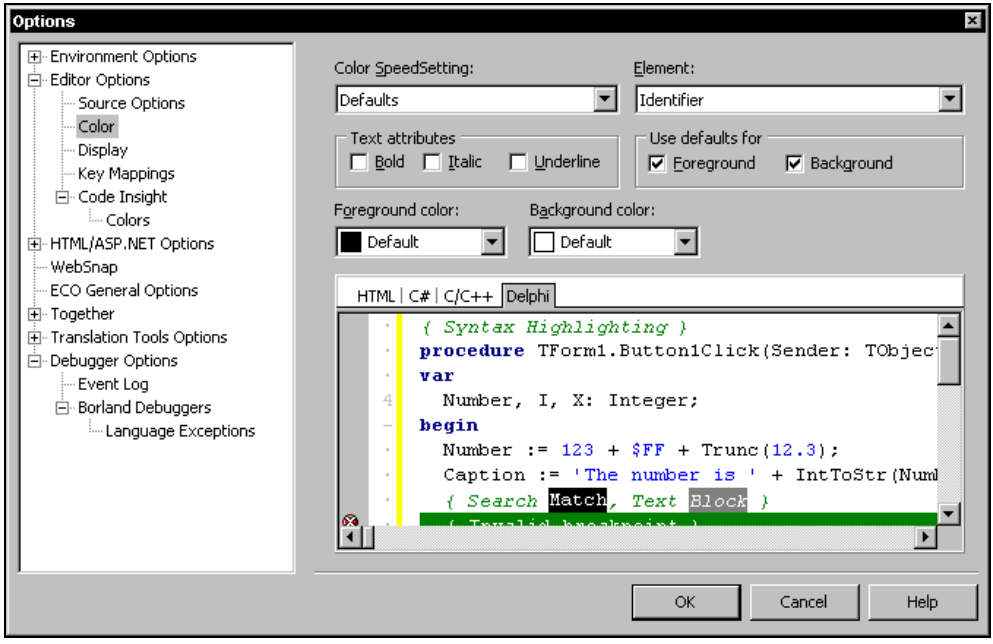


Рис. 3.12. Диалоговое окно **Options**

Система программирования поддерживает технологию *Dock-окон*, которые могут стыковаться (соединяться) друг с другом с помощью мыши. Такими окнами являются инструментальные (не диалоговые) окна интегрированной среды разработки, в том числе окна Менеджера проектов и Проводника данных. Состыкованные окна удобно, например, перемещать по экрану или изменять их размеры.

Для соединения двух окон следует с помощью мыши поместить одно из них на другое и после изменения вида рамки перемещаемого окна отпустить его, после

чего это окно автоматически стыкуется с боком, низом или верхом второго окна. Разделение окон выполняется перемещением состыкованного окна за двойную линию, размещенную под общим заголовком.

Можно запретить стыковку окна, убрав отметку **Dockable** (Стыкуемое) в контекстном меню окна. По умолчанию эта отметка включена и окно является стыкуемым. Стыковку/расстыковку главного окна среды Delphi с остальными ее окнами можно выполнить, задав команду **View** | **Dock Edit Window** (Просмотр | Стыковка окна редактирования).

Для окон Инспектора объектов и Менеджера проектов, а также для Палитры инструментов можно установить режим **Stay on Top** (Расположить наверху), расположив их поверх других окон. Это выполняется включением одноименной отметки в контекстном меню.

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

Характеристика проекта

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

Состав проекта

Создаваемое в среде Delphi .NET 2006 приложение состоит из нескольких файлов, объединенных в *проект*. В состав проекта входят следующие файлы (в скобках указаны расширения имен файлов):

- код проекта (dpr);
- модули и модули форм (pas);
- параметры проекта (dof);
- описания форм для приложения VCL.NET (nfm);
- описания ресурсов (res).

Обычно файлы проекта располагаются в одном каталоге. Поскольку даже относительно простой проект содержит достаточно много файлов, для каждого проекта целесообразно создавать отдельный каталог, где и сохранять все его файлы.

Файл проекта

Файл проекта является центральным файлом проекта и представляет собой собственно программу.

Для приложения Windows Forms файл проекта (без комментариев) содержит следующий код:

```
program Project9;
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.dll'}
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.Data.dll'}
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.Drawing.dll'}
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.Windows.Forms.dll'}
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.XML.dll'}
uses
  System.Reflection,
  System.Runtime.CompilerServices,
  System.Runtime.InteropServices,
  System.Windows.Forms,
  WinForm14 in 'WinForm14.pas' {WinForm14.TWinForm14:
System.Windows.Forms.Form};
{$R *.res}
{$REGION 'Program/Assembly Information'}
[assembly: AssemblyDescription('')]
[assembly: AssemblyConfiguration('')]
[assembly: AssemblyCompany('')]
[assembly: AssemblyProduct('')]
[assembly: AssemblyCopyright('')]
[assembly: AssemblyTrademark('')]
[assembly: AssemblyCulture('')]

[assembly: AssemblyVersion('1.0.*')]

[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile('')]
[assembly: AssemblyKeyName('')]
[assembly: ComVisible(False)]
{$ENDREGION}
[STAThread]
begin
  Application.Run(TWinForm14.Create);
end.
```

Имя проекта (программы) совпадает с именем файла проекта и указывается при сохранении этого файла на диске, в нашем примере это имя `Project9`. То же имя имеют файлы ресурсов и параметров проекта, при переименовании файла проекта данные файлы переименовываются автоматически.

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

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

Для приложения VCL.NET файл проекта содержит следующий код:

```
program Project8;
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.dll'}
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.Data.dll'}
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.Drawing.dll'}
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.XML.dll'}
uses
  System.Reflection,
  System.Runtime.CompilerServices,
  System.Runtime.InteropServices,
  SysUtils,
  Forms,
  Unit2 in 'Unit2.pas' {Form2};
{$R *.res}
{$REGION 'Program/Assembly Information'}
[assembly: AssemblyDescription('')]
[assembly: AssemblyConfiguration('')]
[assembly: AssemblyCompany('')]
[assembly: AssemblyProduct('')]
[assembly: AssemblyCopyright('')]
[assembly: AssemblyTrademark('')]
[assembly: AssemblyCulture('')]

[assembly: AssemblyVersion('1.0.*')]

[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile('')]
[assembly: AssemblyKeyName('')]

[assembly: ComVisible(False)]
//[assembly: Guid('')]
//[assembly: TypeLibVersion(1, 0)]
{$ENDREGION}

[STAThread]
begin
  Application.Initialize;
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

В разделе `uses` здесь 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, в примере это модуль `Unit2` формы `Form2`.

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

Программа проекта для приложения VCL.NET содержит всего три команды, выполняющие инициализацию приложения, создание формы `Form2` и запуск приложения.

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

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

Файл описания формы приложения VCL.NET

Для каждой формы в составе проекта приложения VCL.NET автоматически создаются файл описания формы (расширение nfm) и файл модуля формы (расширение pas).

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

Содержимое файла описания формы определяет ее вид. При необходимости можно отобразить этот файл на экране в текстовом виде, что выполняется командой **View as Text** (Просмотреть как текст) контекстного меню формы. При этом окно Формы пропадает с экрана, а содержимое файла описания формы открывается в окне Редактора кода и доступно для просмотра и редактирования. Для примера ниже приведен текст файла описания простой формы: она содержит метку `Label1`, для которой создан обработчик события `OnClick`.

```
object Form2: TForm2
  Caption = 'Form2'
  ClientHeight = 215
  ClientWidth = 339
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  Left = 0
  Top = 0
  PixelsPerInch = 96
  TextHeight = 13
  object Label1: TLabel
    Caption = 'Label1'
    Left = 32
    Top = 24
    Width = 73
    Height = 33
    OnClick = Label1Click
  end
end
```

Отметим, что в начальной (пустой) форме типа `TForm2` отсутствуют строки, относящиеся к метке `Label1` (выделены курсивом).

Из примера видно, что файл описания содержит перечень всех объектов формы, включая саму форму, а также свойства этих объектов. Для каждого объекта указывается его тип; для формы ее тип (класс) `TForm2` описывается в модуле этой формы. Если в строчке `Caption = 'Form2'`, определяющей заголовок формы, вместо `Form2` ввести, например, текст `Первая форма`, то заголовок формы изменится на новый. На практике такие действия обычно выполняются в окне Инспектора объектов.

Вернуться к режиму конструирования формы можно с помощью команды **View as Form** (Просмотреть как форму) контекстного меню Редактора кода с текстом описания формы.

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

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

Для приложения Windows Forms файл описания формы, в отличие от приложения VCL.NET, отдельно не создается. Описание формы в этом случае входит в состав модуля формы.

Файлы модуля формы

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

```
unit Unit2;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs;
type
  TForm2 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form2: TForm2;
implementation
{$R *.nfm}
end.
```

Delphi .NET 2006 автоматически создает файл модуля формы при добавлении новой формы. К проекту приложения VCL.NET по умолчанию добавляется новая форма типа `TForm`, не содержащая компонентов.

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

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

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

Для пустой формы, добавляемой к проекту *приложения Windows Worms* по умолчанию, файл модуля формы (без комментариев) содержит следующий код:

```
  {$REGION 'Designer Managed Code'}
  strict private
    Components: System.ComponentModel.Container;
    procedure InitializeComponent;
  {$ENDREGION}
  strict protected
    procedure Dispose(Disposing: Boolean); override;
  private
    { Private Declarations }
  public
    constructor Create;
  end;

  [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm13))]

implementation

{$AUTOBOX ON}
procedure TWinForm13.InitializeComponent;
begin
  Self.Components := System.ComponentModel.Container.Create;
  Self.Size := System.Drawing.Size.Create(300, 300);
  Self.Text := 'WinForm13';
end;
{$ENDREGION}

procedure TWinForm13.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
  inherited Dispose(Disposing);
end;
constructor TWinForm13.Create;
begin
  inherited Create;
  InitializeComponent;
end;

end.
```

Замечание

При компиляции модуля автоматически создается файл с расширением dcuil (dcu — для платформы Win32), который содержит откомпилированный код модуля. Этот файл можно удалить из каталога, в котором находятся все файлы проекта, — Delphi снова создаст его при следующей компиляции модуля или проекта. Смысл создания названных файлов в Delphi .NET состоит в том, что практически можно собрать проект, используя только файлы dcuil без файлов pas.

Файлы модулей

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

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

Файл ресурсов

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

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

Параметры проекта

Для установки параметров проекта используется окно параметров проекта (**Project Options**), открываемое командой **Project** | **Options** (Проект | Параметры) или нажатием комбинации клавиш <Shift>+<Ctrl>+<F11>. Параметры разбиты на группы, каждая из которых располагается в окне параметров проекта на своей странице. Например, в группе **Compiler** (Компилятор) можно установить параметры, показанные на рис. 3.13.

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

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

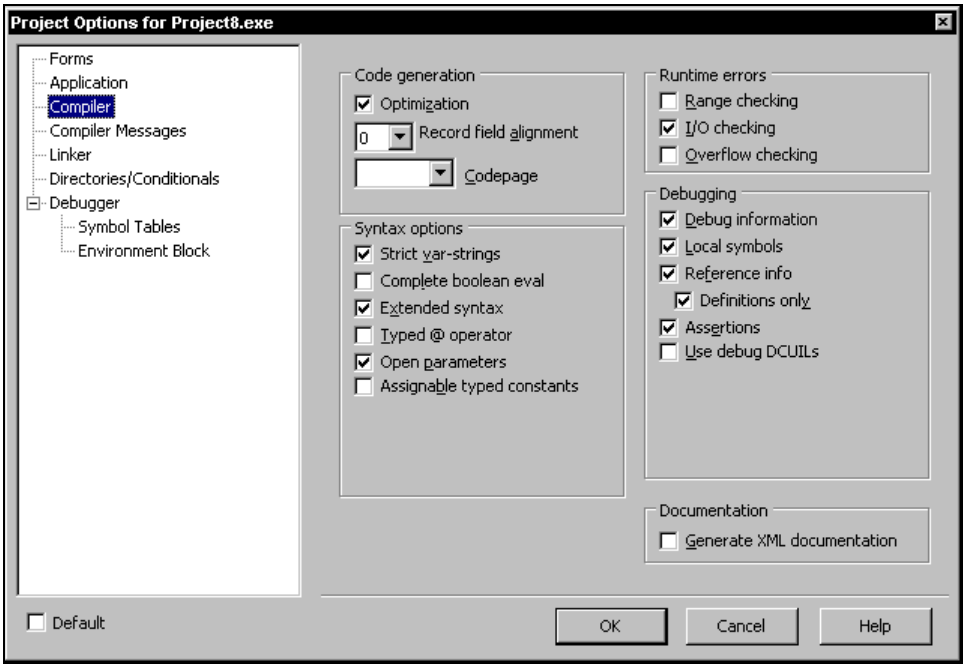


Рис. 3.13. Окно параметров проекта группы **Compiler**

Компиляция и выполнение проекта

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

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

Скомпилировать проект для получения приложения можно на любой стадии разработки проекта. Это удобно для проверки вида и правильности функционирования отдельных компонентов формы, а также для тестирования фрагментов создаваемого кода. При компиляции проекта Delphi .NET 2006 выполняются действия, приведенные ниже.

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

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

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

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

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

Для отладки приложений в среде Delphi можно использовать средства отладчика.

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

Разработка приложения

Borland Developer Studio 2006 относится к *системам визуального программирования*, называемым также *системами RAD* (Rapid Application Development, быстрая разработка приложений).

Разработка приложения в Borland Developer Studio 2006 включает два взаимосвязанных этапа:

- создание пользовательского интерфейса приложения;
- определение функциональности приложения.

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

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

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

Простейшее приложение

Создадим для примера простейшее приложение. Для создания приложения Windows Forms в Палитре инструментов выберем вариант **Windows Forms Application**, а для создания приложения VCL.NET — выберем вариант **VCL Forms Application**. Можно также воспользоваться соответствующей командой, например: **File | New | Windows Forms Application — Delphi for NET**. Заметим, что в данном случае создавать и тем более программировать не придется вообще ничего — Borland Developer Studio 2006 предоставляет готовое приложение, состоящее из одной формы.

Сразу после создания нового приложения Borland Developer Studio 2006 предлагает разработчику "пустую" форму. Данная форма не является пустой в буквальном смысле слова — она содержит основные элементы окна Windows: заголовок **Form1** (для приложения VCL.NET) или **WinForm11** (для приложения Windows Forms), кнопки сворачивания, разворачивания и закрытия окна, изменения размеров окна и кнопку вызова системного меню окна.

Любое приложение Windows выполняется в соответствующем окне. Даже если оно не имеет функциональности, т. е. является пустым, то все равно должно иметь свое окно. В Delphi .NET 2006 для любого разрабатываемого приложения VCL.NET автоматически предлагается окно (форма), для которого уже созданы два файла — с описанием и модулем. Для любого разрабатываемого приложения Windows Forms автоматически предлагается окно, для которого уже создан один файл с модулем, в котором содержится в том числе и описание окна.

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

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

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

Замечание

При компиляции проекта можно использовать специальные пакеты динамически загружаемых библиотек (DLL), что позволяет существенно уменьшить размер приложения. Однако при этом приложение уже не является автономным и в процессе своей работы обращается к пакетам, которые были задействованы при компиляции проекта.

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

Создание интерфейса приложения

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

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

К *визуальным компонентам* относятся, например, кнопки, списки или переключатели и сама форма. Так как с помощью визуальных компонентов пользователь

управляет приложением, их также называют *управляющими компонентами* или *элементами управления*. Именно визуальные компоненты образуют пользовательский интерфейс приложения.

К *невизуальным компонентам* относятся компоненты, выполняющие важные вспомогательные действия, не связанные с непосредственным управлением приложением и не требующие своего отображения на форме при выполнении приложения. К невизуальным компонентам относятся, например, таймер (компонент типа `TTimer` в приложениях VCL.NET, компонент типа `Timer` в приложениях Windows Forms), источник данных (компонент типа `TDataSource` в приложениях VCL.NET) и набор данных (компонент типа `DataSet` в приложениях Windows Forms).

Замечание

Напомним, что при создании приложения Windows Forms *невизуальные* компоненты размещаются на индикаторе компонентов Конструктора формы под окном формы. При создании приложения VCL.NET *невизуальные* компоненты размещаются на самой форме.

При создании интерфейса приложения для каждого компонента выполняются следующие операции:

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

Разработчик выполняет эти операции в окне Формы, используя Палитру инструментов 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 программирование. В связи с этим часто работу по созданию интерфейса называют конструированием.

Выбор компонента в Палитре инструментов выполняется щелчком мыши на нужном компоненте, например, на кнопке `Button`. Если после этого щелкнуть на свободном месте формы, то на ней появляется выбранный компонент. Значки компонентов отражают назначение компонентов. Кроме того, при наведении на компонент указателя мыши отображается всплывающая подсказка с информацией о его пространстве имен.

Замечание

Обозначения типов объектов в Delphi .NET 2006 для приложений VCL Forms, в том числе и компонентов, начинаются с буквы `T`. Иногда типы (`TButton`, `TLabel`) используются вместо имен (`Button`, `Label`) для обозначения компонентов. Для приложений Windows Forms типы объектов и компонентов обозначают без буквы `T`, например, `Button`, `Label`. Имена соответствующих этим типам объектов и компонентов совпадают с именами типов, но при этом содержат номер, например, `Button1`, `Label2`.

После размещения компонента на форме *приложения VCL.NET* система Delphi .NET 2006 автоматически вносит изменения в файлы модуля и описания формы. В описание класса формы (файл модуля формы) для каждого нового компонента добавляется строчка формата

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

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

```
Button1: TButton;
```

После размещения компонента на форме *приложения Windows Forms* система Delphi .NET 2006 автоматически вносит изменения в файл модуля. Так, для кнопки типа `Button` в файле модуля формы в разделе `interface` будет добавлена строка

```
Button1: System.Windows.Forms.Button;
```

и в теле процедуры `TWinForm13.InitializeComponent` из раздела `implementation` будут добавлены строки:

```
Self.Button1 := System.Windows.Forms.Button.Create;
  Self.SuspendLayout;
  //
  // Button1
  //
  Self.Button1.Location := System.Drawing.Point.Create(72, 80);
  Self.Button1.Name := 'Button1';
  Self.Button1.Size := System.Drawing.Size.Create(40, 40);
  Self.Button1.TabIndex := 0;
  Self.Button1.Text := 'Button1';
```

После размещения компонента на форме можно с помощью мыши изменять его положение и размеры. В случае нескольких компонентов можно выполнять выравнивание или перевод того или иного компонента на передний или задний план с помощью команд контекстного меню компонентов.

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

командой меню **Tools** (Средства). Для приложений VCL Forms в окне параметров открывается группа **VCL Designer** (Конструктор VCL), а для приложений Windows Forms — группа **Windows Forms Designer** (Конструктор Windows Forms).

Внешний вид компонента определяется его свойствами, которые становятся доступными в окне Инспектора объектов, когда компонент выделен на форме и вокруг него появились маркеры выделения (рис. 3.14). Доступ к свойствам самой формы осуществляется аналогично, однако в выбранном состоянии форма не выделяется маркерами. Для выделения (выбора) формы достаточно щелкнуть в любом ее месте, свободном от других компонентов.

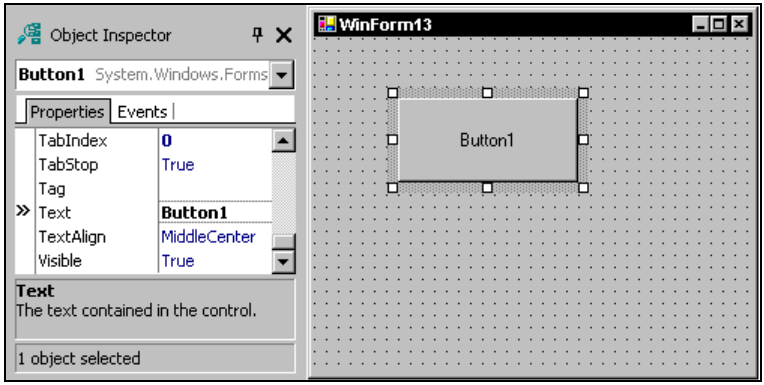


Рис. 3.14. Доступ к свойствам компонента

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

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

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

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

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

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

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

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

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

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

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

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

- *простой* (текстовый) — значение свойства вводится или редактируется как обычная строка символов, которая интерпретируется как числовой или строковый тип; используется для таких свойств, как `Caption`, `Text`, `Left` и `Hint`;

- *перечисляемый* — значение свойства выбирается в раскрывающемся списке. Список раскрывается щелчком на стрелке, которая появляется при установке указателя мыши в области значения свойства. Используется для таких свойств, как `FormStyle`, `Visible` и `ModalResult`;
- *множественный* — значение свойства представляет собой комбинацию значений из предлагаемого множества. В окне Инспектора объектов слева от имени свойства множественного типа стоит знак "+". Формирование значения свойства выполняется с помощью дополнительного списка, раскрываемого двойным щелчком на имени свойства. Этот список содержит перечень всех допустимых значений свойства, справа от каждого значения можно указать `True` или `False`. Выбор `True` означает, что данное значение включается в комбинацию значений, а `False` — нет. Используется для таких свойств, как `BorderIcons` и `Anchors`;
- *объекта* — свойство является объектом и, в свою очередь, содержит другие свойства (подсвойства), каждое из которых можно редактировать отдельно. Используется для таких свойств, как `Font`, `Items` и `Lines`. В области значения свойства-объекта в скобках указывается тип объекта, например, (`TFont`) или (`TSrings`). Для свойства-объекта слева от имени может стоять знак "+", в этом случае управление его свойствами выполняется, как и для свойства множественного типа, через раскрывающийся список. Этот список в левой части содержит имена подсвойств, а в правой — значения, редактируемые обычным способом. В области значения может отображаться кнопка с тремя точками. Это означает, что для данного свойства имеется специальный редактор, который вызывается нажатием этой кнопки. Так, для свойства `Font` открывается стандартное окно Windows для установки параметров шрифта.

При *выполнении* приложения значения свойств компонентов (доступных в окне Инспектора объектов) можно изменять с помощью инструкций присваивания, например, в обработчике события создания формы. Так, изменение заголовка метки `Label1` можно выполнить следующим образом:

```
Label1.Text:='Метка1';
```

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

Отметим, что есть свойства времени выполнения, недоступные через Инспектор объектов, с которыми можно работать только во время выполнения приложения. К таким свойствам относится, например, число записей `RecordCount` набора данных.

Определение функциональности приложения

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

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

Допустим, что при создании интерфейса приложения в форме была размещена кнопка `Button`, предназначенная для закрытия окна. По умолчанию эта кнопка получила имя и заголовок `Button1`, однако заголовок (свойство `Caption` или `Text`) посредством окна Инспектора объектов был заменен более осмысленным — `Выход`. При выполнении приложения кнопку `Button1` можно нажимать с помощью мыши или клавиатуры. Кнопка отображает нажатие визуально, однако никаких действий, связанных с закрытием формы, не выполняется. Это происходит потому, что для кнопки не определена реакция на какие-либо действия пользователя, в том числе и на ее нажатие.

Чтобы кнопка могла реагировать на то или иное событие, для него необходимо создать или указать процедуру обработки, которая будет вызываться при возникновении этого события. Для создания процедуры обработки события, или обработчика, нужно выделить в форме кнопку и перейти на страницу **Events** (События) окна Инспектора объектов, где указываются все возможные для данной кнопки события (рис. 3.15).

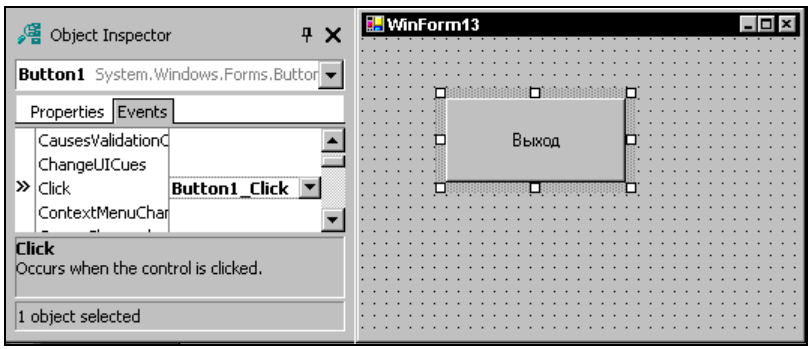


Рис. 3.15. Доступ к событиям компонента

При нажатии кнопки возникает событие `Click`, поэтому следует создать обработчик именно этого события. Нужно сделать двойной щелчок в области значения события `Click`, в результате Delphi автоматически создает в модуле формы заготовку процедуры-обработчика. При этом окно Редактора кода переводится на передний план, а курсор устанавливается в то место процедуры, где программист должен написать код, выполняемый при нажатии кнопки `Button1`. Поскольку кнопка должна закрывать окно, в этом месте можно указать `Form1.Close` (для приложения VCL Forms), `TWinForm13.ActiveForm.Close;` (для приложения Windows Forms) или просто `Close`.

Так, в модуле формы приложения Windows Forms процедура обработки нашего события будет иметь следующий вид:

```
procedure TWinForm13.Button1_Click(sender: System.Object; e: System.EventArgs);
begin
TWinForm13.ActiveForm.Close;
end;
```

Здесь полужирным начертанием выделен код, набранный программистом; заголовок процедуры-обработчика среда Delphi создала автоматически.

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. Имя выбранного свойства или метода автоматически добавляется справа от точки. Если метод содержит параметры, то отображается подсказка, содержащая состав и типы параметров.

Перечень стандартных конструкций языка вызывается нажатием комбинации клавиш <Ctrl>+<J>. После выбора требуемой конструкции автоматически добавляется ее код. Например, при выборе оператора `for` в модуле появится следующий код:

```
for I := 0 to List.Count — 1 do
```

Имя обработчика `TWinForm13.Button1_Click` или `TForm1.Button1Click` (для приложения VCL.NET) образуется прибавлением к имени компонента имени события без префикса `On`. Это имя является значением события, для которого создан обработчик, в нашем случае — для события нажатия кнопки `OnClick`. При изменении через окно Инспектора объектов имени кнопки происходит автоматически переименование этой процедуры в соответствующих файлах PAS и NFM (для приложения VCL.NET) проекта.

Аналогично создаются обработчики для других событий и других компонентов. Для удаления процедуры-обработчика достаточно удалить код, который про-

граммист внес в нее самостоятельно. После этого при сохранении или компиляции модуля обработчик будет удален автоматически из всех файлов проекта.

Замечание

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

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

Одну и ту же процедуру можно связать с несколькими событиями, в том числе для различных компонентов. Такая процедура называется *общим (разделяемым) обработчиком* и вызывается при возникновении любого из связанных с ней событий.

Встроенный отладчик

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

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

Установка параметров отладчика выполняется на странице группы **Debugger Options** (Параметры отладчика) в диалоговом окне **Options** (Параметры), открываемом командой **Tools | Options** (Сервис | Параметры) (рис. 3.16).

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

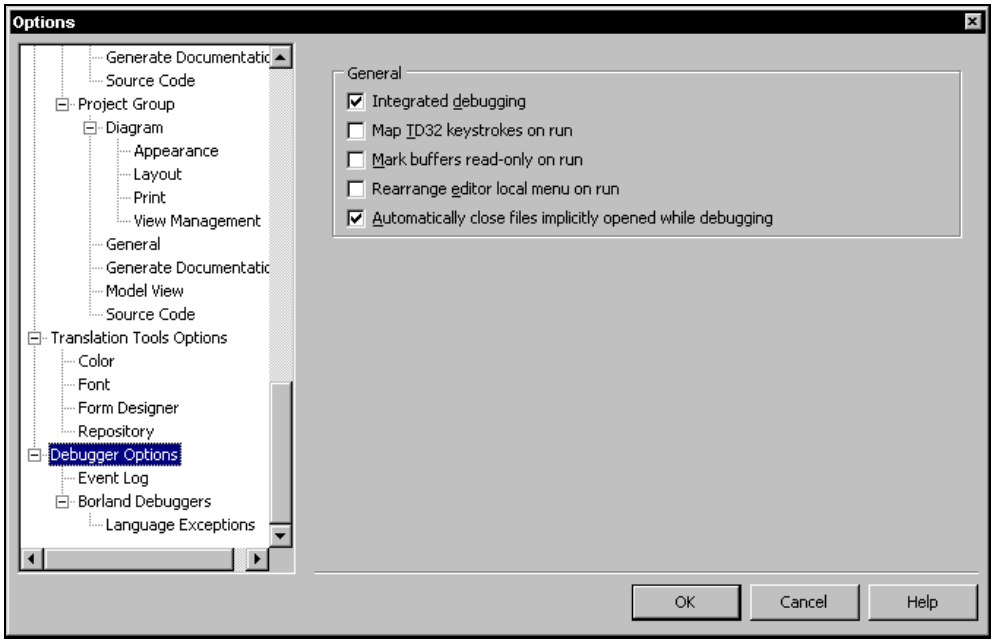


Рис. 3.16. Окно установки параметров отладчика

Рефакторинг

В системе Delphi .NET 2006 имеется технология улучшения характеристик исходного кода приложения, называемая *рефакторинг* (Refactoring). Рефакторинг выполняется с помощью команд меню **Refactor** (Рефактор). При выполнении рефакторинга можно изменить структуру исходного кода приложения или модифицировать его таким образом, чтобы функциональность приложения сохранилась, и при этом улучшились характеристики производительности и надежности. В числе операций рефакторинга можно, к примеру, назвать символьное переименование, извлечение метода, поиск модуля, поиск ссылок и др.

Справочная система

Справочная система Delphi .NET 2006 состоит из стандартной системы справки, справочной помощи через Интернет и контекстно-зависимой справочной помощи.

Стандартная система справки вызывается командой **Borland Help** (Помощь Borland) меню **Help** (Помощь) или с помощью кнопки **Help** (Помощь) страницы приглашения (Welcome Page). При этом открывается окно навигатора справочной системы **Borland Help — Welcome**.

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

Вкладка **Index** (Предметный указатель) окна навигатора справочной системы позволяет выполнить поиск нужной информации с помощью указателя, ключевые слова в котором расположены в алфавитном порядке. Чтобы найти нужную справку, достаточно указать первые несколько букв ключевого слова в поле **1** вкладки или выбрать нужное слово в списке поля **2** и выполнить на нем щелчок мышью или нажать кнопку **Display** (Вывести). Пользоваться этой вкладкой проще, чем вкладкой **Contents**, поскольку при поиске справочной информации пользователь не обязан знать, к какому разделу оглавления относится интересующий его вопрос, ему достаточно указать соответствующее ключевое слово.

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

Вкладка **Favorites** (Предпочтения) содержит набор ссылок на популярные сайты Интернета.

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

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

ЧАСТЬ II

Приложения Windows Forms

ГЛАВА 4

Характеристика управляющих компонентов

Напомним, что при разработке приложений Windows Forms с помощью системы программирования Delphi .NET 2006 используется "родная" для платформы .NET библиотека FCL (Framework Class Library) — библиотека классов .NET Framework. Достоинством создания таких приложений, кроме всего прочего, является то, что при этом применяются средства, привычные для популярных систем программирования фирмы Microsoft — Visual C .NET, Visual C# .NET и Visual Basic .NET, которыми пользуется широкий круг программистов.

Компоненты библиотеки классов .NET Framework во многом повторяют основные свойства компонентов библиотеки VCL.NET. Заметим, что в приложениях Windows Forms имена компонентов (они же задают имена классов) в Палитре инструментов и имена создаваемых с их помощью объектов практически одинаковы. Например, компонент `Label` (надпись) при помещении его на форму получает имя `LabelN`, где `N` — номер по порядку объекта, созданного с помощью этого компонента.

В библиотеке VCL.NET имена компонентов (они же имена классов) начинаются с символа `T`, в имени создаваемого объекта этот символ отсутствует. Например, компонент `TButton` (кнопка) при помещении его на форму получает имя `ButtonN`, где `N` — также номер по порядку объекта, созданного с помощью этого компонента.

Страницы с компонентами

Для создания интерфейса приложений Windows Forms система Delphi .NET 2006 предоставляет широкий набор управляющих компонентов, основные из которых располагаются на страницах **Windows Forms**, **Components** и **Dialogs** Палитры инструментов. Большинство компонентов страницы **Windows Forms** являются визуальными, для страниц **Components** и **Dialogs** большинство компонентов яв-

ляются не визуальными. Все компоненты указанных страниц принадлежат пространству имен `System.Windows.Forms`.

На странице **Windows Forms** (рис. 4.1) Палитры инструментов находятся следующие управляющие компоненты:

- `Label` (надпись);
- `LinkLabel` (гиперссылка);
- `Button` (стандартная кнопка);
- `TextBox` (текстовое поле);
- `Panel` (панель);
- `CheckBox` (независимый переключатель, флажок);
- `RadioButton` (зависимый переключатель, переключатель);
- `ComboBox` (поле со списком);
- `ListBox` (список);
- `CheckedListBox` (список с флажком);
- `TreeView` (представление дерева);
- `ListView` (представление списка);
- `TabControl` (вкладка);
- `PictureBox` (поле с рисунком);
- `Splitter` (разделитель);
- `ToolBar` (панель инструментов);
- `MonthCalendar` (календарь);
- `DateTimePicker` (выбор даты и времени);
- `TrackBar` (ползунок);
- `VScrollBar` (вертикальная полоса прокрутки);
- `HScrollBar` (горизонтальная полоса прокрутки);
- `NumericUpDown` (числовой счетчик);
- `DomainUpDown` (строковый счетчик);
- `GroupBox` (группа);
- `PropertyGrid` (свойство сетки);
- `StatusBar` (строка состояния);
- `PrintPreviewControl` (предварительный просмотр);
- `RichTextBox` (расширенный текстовый редактор);

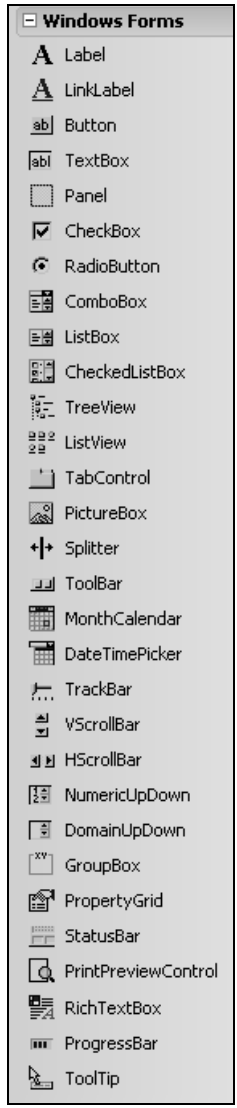


Рис. 4.1. Страница **Windows Forms** Палитры инструментов

- `ProgressBar` (индикатор процесса);
- `ToolTip` (всплывающая подсказка).

На странице **Components** (рис. 4.2) Палитры инструментов находятся следующие управляющие компоненты:

- `MainMenu` (главное меню);
- `ContextMenu` (контекстное меню);
- `NotifyIcon` (значок уведомления);
- `PrintDocument` (печать документа);
- `ImageList` (список изображений);
- `Timer` (таймер);
- `ErrorProvider` (провайдер ошибок);
- `HelpProvider` (провайдер справочной помощи).

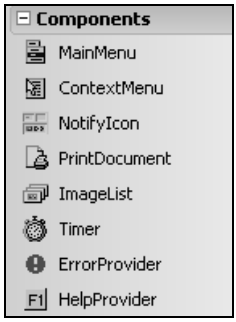


Рис. 4.2. Страница **Components**

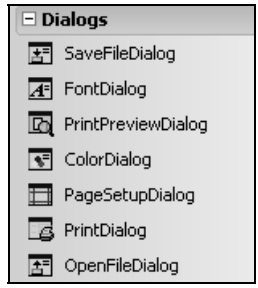


Рис. 4.3. Страница **Dialogs**

На странице **Dialogs** (рис. 4.3) Палитры инструментов находятся компоненты, с помощью которых обеспечивается создание стандартных диалоговых окон:

- `SaveFileDialog` (сохранение файла);
- `FontDialog` (настройка параметров шрифта);
- `PrintPreviewDialog` (просмотр перед печатью);
- `ColorDialog` (выбор цвета);
- `PageSetupDialog` (настройка параметров страницы);
- `PrintDialog` (настройка печати);
- `OpenFileDialog` (открытие файла).

Общие свойства визуальных компонентов

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

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

- доступность (**Accessibility**);
- вид (**Appearance**);
- поведение (**Behavior**);
- конфигурация (**Configurations**);
- данные (**Data**);
- конструирование (**Design**);
- фокус (**Focus**);
- макет (**Layout**);
- разное (**Miscellaneous**).

Родительским классом всех интерфейсных компонентов, имеющих или не имеющих визуальное представление, является класс `Component`, принадлежащий пространству имен `System.ComponentModel`. Все компоненты, имеющие визуальное представление, наследуются от класса `Control`, принадлежащего пространству имен `System.Windows.Forms` и непосредственно наследуемого от класса `Component`.

В табл. 4.1 приведены некоторые важные свойства визуальных компонентов, унаследованные от класса `Control`.

Таблица 4.1. *Свойства визуальных компонентов*

Свойство	Тип свойства	Назначение
`AllowDrop`	`Boolean`	Получает или устанавливает значение, указывающее, может ли компонент принимать данные, передаваемые ему путем перетаскивания (методом Drag and Drop)
`BackColor`	`Color`	Получает или устанавливает цвет фона компонента
`BackGroundImage`	`Image`	Получает или устанавливает фоновый рисунок

Таблица 4.1 *(окончание)*

Свойство	Тип свойства	Назначение
Bounds	Rectangle	Получает или устанавливает размеры и положение компонента вместе с его клиентской областью
CanFocus	Boolean	Получает значение, указывающее допустимость получения фокуса компонентом
CanSelect	Boolean	Получает значение, указывающее допустимость выделения компонента
ClientRectangle	Rectangle	Получает прямоугольник, представляющий клиентскую область компонента
ClientSize	Size	Получает или устанавливает размеры клиентской области компонента
ContextMenu	ContextMenu	Получает или устанавливает контекстное меню, связанное с компонентом
Focused	Boolean	Получает значение, указывающее, имеет ли компонент фокус ввода (True, если компонент содержит фокус ввода)
Font	Font	Получает или устанавливает шрифт, используемый для отображения текста
ForeColor	Color	Получает или устанавливает цвет текста в поле компонента
Height	Integer	Получает или устанавливает высоту компонента в пикселах
Name	string	Получает или устанавливает имя компонента, используемое при его указании
Visible	Boolean	Получает или устанавливает значение, указывающее видимость компонента
Width	Integer	Получает или устанавливает ширину компонента в пикселах

Замечание

Компоненты Panel, GroupBox, PictureBox, ProgressBar, Splitter, Label и LinkLabel (при отсутствии гиперссылки в элементе управления) не могут быть выделенными, и их свойство CanSelect возвращает значение False.

Общие методы визуальных компонентов

С визуальными компонентами, как и с другими объектами, связано большое количество методов, позволяющих создавать и удалять объекты, прорисовывать их, отображать и скрывать, а также выполнять другие операции. Как отмечалось,

общие методы визуальных компонентов происходят от класса `Control`. В табл. 4.2 приведены некоторые, на наш взгляд, важнейшие из этих методов.

Таблица 4.2. *Общие методы визуальных компонентов*

Метод	Назначение
`procedure BringToFront;`	Перемещает компонент на слой ближе к фронту в z-порядке
`procedure CreateControl`	Вызывает создание компонента, включая создание дескриптора и дочерних элементов
`procedure Dispose;`	Перегружаемый метод. Освобождает все ресурсы, используемые компонентом
`function DoDragDrop(data: TObject; allowedeffects: DragDropEffects): DragDropEffects`	Начинает операцию перетаскивания методом Drag and Drop
`function FindForm: Form`	Возвращает форму, на которой размещен компонент
`function Focus: Boolean`	Устанавливает фокус ввода на компоненте и возвращает значение `True` в случае успеха
`procedure Hide;`	Скрывает компонент
`function IsInputKey(K: keys): Boolean;`	Определяет, является ли заданная клавиша обычной входной клавишей или специальной клавишей, требующей препроцессорной обработки
`function PointToClient(p Point): Point;`	Преобразует экранные координаты точки в координаты клиентской области компонента
`function PointToScreen(p Point): Point;`	Преобразует координаты точки в клиентской области компонента в экранные координаты
`function RectangleToClient(r Rectangle): Rectangle;`	Преобразует размеры и положение прямоугольника на экране в клиентские координаты
`function RectangleToScreen(r Rectangle): Rectangle;`	Преобразует размеры и положение прямоугольника в клиентской области в экранные координаты
`procedure Show;`	Устанавливает отображение компонента

В качестве примера приведем код, в котором содержится вызов метода `Focus`:

```
If TextBox1.CanFocus then TextBox1.Focus;
```

Здесь перед получением компонентом `TextBox1` фокуса ввода производится проверка возможности передачи ему фокуса.

Общие события визуальных компонентов

Визуальные компоненты способны генерировать и обрабатывать большое число (несколько десятков) событий различных видов. К наиболее часто имеющим место событиям можно отнести следующие:

- выбор элемента управления;
- перемещение указателя мыши;
- вращение колеса мыши;
- нажатие клавиш клавиатуры;
- получение и потеря элементом управления фокуса ввода;
- перемещение объектов методом Drag and Drop (перетаскиванием).

Отметим, что в окне Инспектора объектов события объединены в следующие группы:

- действие (**Action**);
- поведение (**Behavior**);
- данные (**Data**);
- перетаскивание (**Drag Drop**);
- фокус (**Focus**);
- клавиша (**Key**);
- разметка (**Layout**);
- мышь (**Mous**);
- изменение свойства (**Property Changed**).

В табл. 4.3 приведены общие (наследуемые от класса `Control`) события для визуальных компонентов, возникающие при работе с ними наиболее часто.

Таблица 4.3. *Общие события визуальных компонентов*

Событие	Условие возникновения
`Click: EventHandler;`	Щелчок на компоненте
`DoubleClick: EventHandler;`	Двойной щелчок на компоненте
`DragDrop: DragEventHandler;`	Завершение операции перетаскивания
`DragEnter: DragEventHandler;`	Вхождение перетаскиваемого объекта в границы компонента

Таблица 4.3 *(окончание)*

Событие	Условие возникновения
`DragLeave: DragEventHandler;`	Выход перетаскиваемого объекта за пределы границ компонента
`DragOver: DragEventHandler;`	Перетаскивание объекта над границами компонента
`Enter: EventHandler;`	Появление указателя мыши в области компонента
`GotFocus: EventHandler;`	Получение фокуса компонентом
`KeyDown: KeyEventHandler;`	Нажатие любой клавиши, когда компонент имеет фокус ввода
`KeyPress: KeyPressEventHandler;`	Нажатие алфавитно-цифровой клавиши, когда компонент имеет фокус ввода
`KeyUp: KeyEventHandler;`	Отпускание клавиши, когда компонент имеет фокус ввода
`Leave: EventHandler;`	Покидание фокуса ввода компонентом
`LostFocus: EventHandler;`	Потеря фокуса ввода компонентом
`MouseDown: MouseEventHandler;`	Нажатие кнопки мыши при размещении указателя над компонентом
`MouseEnter: MouseEventHandler;`	Появление указателя мыши над компонентом
`MouseHover: MouseEventHandler;`	Указатель мыши находится над компонентом
`MouseLeave: MouseEventHandler;`	Указатель мыши покидает компонент
`MouseMove: MouseEventHandler;`	Указатель мыши перемещается над компонентом
`MouseUp: MouseEventHandler;`	Отпускание кнопки мыши
`MouseWheel: MouseEventHandler;`	Поворачивание колеса мыши
`Paint: PaintEventHandler;`	Отрисовка компонента
`Resize: EventHandler;`	Изменение размеров компонента
`TextChanged: EventHandler;`	Изменение значения свойства `Text`
`Validated: EventHandler;`	Завершение подтверждения события компонента
`Validating: CancelEventHandler;`	Выполнение подтверждения события компонента

Замечание

Не все из приведенных нами общих событий визуальных компонентов содержатся в Инспекторе объектов для различных визуальных компонентов. Например, для компонента `Button` в Инспекторе объектов содержится событие `Click`, но отсутствует событие `DoubleClick`.

Указанные в табл. 4.3 события, связанные с получением и потерей фокуса ввода, возникают в следующем порядке: `Enter`, `GotFocus`, `Leave`, `Validating`, `Validated` и `LostFocus`. При этом если свойство `CausesValidation` имеет значение `False`, то события `Validating` и `Validated` подавляются.

События, связанные с использованием клавиатуры, возникают в следующем порядке: `KeyDown`, `KeyPress` и `KeyUp`.

События, связанные с применением мыши, возникают в следующем порядке: `MouseEnter`, `MouseMove`, `MouseHover`/`MouseDown`/`MouseWheel`, `MouseUp` и `MouseLeave`.

Обработка событий клавиатуры

События, связанные с использованием клавиатуры (см. табл. 4.3), можно обрабатывать с целью управления приложением.

Обработчик событий `KeyDown`, `KeyPress` и `KeyUp` клавиатуры получает аргументы в виде данных типа `KeyEventArgs`, связанных с соответствующим событием. При этом доступны для использования следующие свойства `KeyEventArgs`:

- `Alt` — получает значение, указывающее, была ли нажата клавиша <Alt>;
- `Control` — получает значение, указывающее, была ли нажата клавиша <Ctrl>;
- `Handled` — получает или устанавливает значение, указывающее, выполнялась ли обработка события;
- `KeyCode` — получает код клавиатуры для события `KeyDown` или `KeyUp`;
- `KeyData` — получает данные клавиатуры для события `KeyDown` или `KeyUp`;
- `KeyValue` — получает значение клавиатуры для события `KeyDown` или `KeyUp`;
- `Modifiers` — получает флаги модификации для события `KeyDown` или `KeyUp`. Это указывает на то, какая комбинация управляющих клавиш (<Ctrl>, <Shift> и <Alt>) была нажата;
- `Shift` — получает значение, указывающее, была ли нажата клавиша <Shift>.

Для примера рассмотрим обработчик события `KeyDown` для компонента `TextBox1`, размещенного на форме, вид которой показан на рис. 4.4. Код нажатой клавиши отображается с помощью свойства `Text` компонента `Label1`.

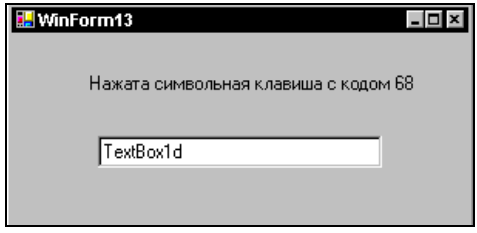


Рис. 4.4. Вид формы с обработкой события `KeyDown`

Например, обработчик события KeyDown может содержать следующий код:

```
procedure TWinForm13.TextBox2_KeyDown(sender: System.Object;
e: System.Windows.Forms.KeyEventArgs);
begin
if e.Modifiers=(Keys.Control or Keys.Shift) then Label1.Text:='Нажата
комбинация  <Shift+Ctrl>'
   else if e.Alt then  Label1.Text:='Нажата клавиша <Alt>'
   else if e.Control then  Label1.Text:='Нажата клавиша <Ctrl>'
   else if e.Shift   then  Label1.Text:='Нажата клавиша <Shift>'
   else Label1.Text:='Нажата  клавиша с кодом '+e.KeyValue.ToString;
end;
```

Еще один вариант обработчика события KeyDown может иметь следующий код:

```
procedure TWinForm13.TextBox1_KeyDown(sender: System.Object;
e: System.Windows.Forms.KeyEventArgs);
begin
case e.KeyValue of
16:  Label1.Text:='Нажата клавиша <Shift>';
17:  Label1.Text:='Нажата клавиша <Ctrl>';
18:  Label1.Text:='Нажата клавиша <Alt>';
else  Label1.Text:='Нажата символьная клавиша с кодом '+ e.KeyValue.ToString;
end;
end;
```

В первом варианте обработчика нами использовано свойство Modifiers, с помощью которого выявляется нажатие комбинации клавиш <Shift>+<Ctrl>. Значение этого свойства определяется с использованием логической операции ИЛИ (or).

Можно задавать обработчики событий клавиатуры для формы и для элементов управления. Причем если свойству формы KeyPreview установить значение True, то сначала будет выполняться обработчик события клавиатуры для формы, а затем — для элемента управления, которому принадлежит фокус. При этом можно запретить обработку события клавиатуры для элемента управления. Для этого следует установить значение True свойству KeyPressEventArgs.Handled в обработчике события KeyPress формы.

Обработка событий мыши

События, связанные с использованием мыши (см. табл. 4.3), можно обрабатывать с целью управления приложением. Обработчик таких событий получает аргументы в виде данных типа MouseEventArgs, связанных с соответствующим событием. При этом доступны для использования следующие свойства MouseEventArgs:

- Button — указывает, какая кнопка мыши была нажата (возможны значения Left, Right, Middle и None);
- Clicks — указывает, какое число раз мышь была нажата и отпущена;

- `Delta` — указывает количество щелчков по зубчикам колесика, имевших место при прокрутке колесика мыши;
- `X` — возвращает X-координату указателя мыши относительно компонента;
- `Y` — возвращает Y-координату указателя мыши относительно компонента.

Приведем примеры возможных обработчиков событий управления мышью. Например, отображение текущих координат указателя мыши над кнопкой (рис. 4.5) можно задать с помощью двух обработчиков события `MouseMove`.

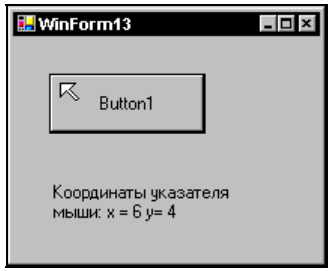


Рис. 4.5. Пример обработки событий управления мышью

Код обработчиков может иметь следующий вид :

```
procedure TWinForm13.Button1_MouseMove(sender: System.Object;
e: System.Windows.Forms.MouseEventArgs);
begin
 label1.Text:='Координаты указателя мыши: x = '+e.X.ToString +
' y='+e.Y.ToString;
end;
procedure TWinForm13.TWinForm13_MouseMove(sender: System.Object;
e: System.Windows.Forms.MouseEventArgs);
begin
   label1.Text:='';
end;
```

Здесь в обработчике события `Button1_MouseMove` (для кнопки) задается отображение координат в свойстве `Text` метки, а в обработчике события `TWinForm13_MouseMove` (для формы) значение этого свойства очищается.

Аналогичным образом могут быть заданы обработчики и других событий, связанных с управлением мышью.

Перетаскивание методом Drag and Drop

Перетаскивание объектов методом Drag and Drop представляет механизм, очень часто применяемый при создании приложений. Используемые при его реализации события приведены в табл. 4.3. С его помощью можно выполнить перетаски-

вание различных объектов с помощью мыши. В качестве перетаскиваемых объектов, к примеру, могут быть текст, его параметры, файлы изображений, элементы управления и др.

Для обеспечения возможности реализации перетаскивания объектов нужно установить для всех компонентов (источников или приемников объектов) их свойству `AllowDrop` значение `True`.

Как следует из табл. 4.2, операция перетаскивания с помощью механизма Drag and Drop начинается путем вызова метода:

```
function DoDragDrop(data: TObject; allowedeffects: DragDropEffects):
DragDropEffects;
```

Параметры этого метода имеют следующее назначение: `data` — перетаскиваемые данные; `allowedeffects` — указывает эффекты от выполнения операции перетаскивания, тип `DragDropEffects` является перечислимым и содержит следующие значения:

- `All` — данные копируются, удаляются из источника и перемещаются в приемник;
- `Copy` — данные копируются из источника в приемник;
- `Link` — данные источника связываются с приемником;
- `Move` — данные из источника перемещаются в приемник;
- `None` — приемник не принимает данные;
- `Scroll` — прокручивание готово начаться или уже выполняется в приемнике.

В результате выполнения рассматриваемого метода он возвращает одно из указанных нами значений типа `DragDropEffects`, который указывает окончательный эффект от выполнения операции перетаскивания.

Метод `DoDragDrop` определяет элемент управления, находящийся в текущее время под курсором. При этом проверяется, может ли этот элемент управления быть приемником для операции перетаскивания. Если да, то вызывается событие `GiveFeedback` с заданным с помощью параметра `allowedeffects` эффектом перетаскивания.

Отслеживаются изменения положения курсора, состояния клавиатуры и состояния кнопок мыши. При этом, если пользователь перемещает указатель за пределы окна, возникает событие `DragLeave`.

Если указатель мыши попадает на другой элемент управления (приемник), то для него возникает событие `DragEnter`. В обработчике этого события для приемника проверяют, соответствует ли формат перетаскиваемых данных тому формату, который ожидает принять этот элемент управления. Если соответствует, то для перетаскиваемого объекта типа `DragEventArgs` его свойству `Effect` задается нужное значение типа `DragDropEffects`. Тем самым обеспечивается возможность завершения операции перетаскивания нужного вида.

При перемещении мыши в пределах того же элемента управления возникает событие `DragOver`.

Если имеет место изменение состояния клавиатуры или кнопок мыши, то возникает событие `QueryContinueDrag` и определяется, нужно ли продолжать операцию перетаскивания, вставлять данные или отменить операцию на основе значения свойства `Action` события `QueryContinueDragEventArgs`.

Если свойство имеет значение `DragAction.Continue`, то возникает событие `DragOver` для продолжения операции и возникает событие `GiveFeedback` с новым эффектом, так что соответствующая визуальная обратная связь могла быть установлена.

События `DragOver` и `GiveFeedback` образуют пару таким образом, что при перемещении мыши через приемник пользователь получает обновляемую обратную связь по отношению к позиции мыши.

В обработчике события `DragDrop`, которое возникает при завершении операции перетаскивания объекта, нужные данные помещаются в элемент управления приемник. Если при перетаскивании объекта указатель мыши перемещается за пределы формы, то операция перетаскивания отменяется в обработчике события `QueryContinueDrag`.

Приведем пример использования механизма Drag and Drop для компонентов, вид которых показан на рис. 4.6. Здесь источниками являются компоненты `ListBox1` (справа вверху) и `Label1` (слева вверху), приемником является компонент `TextBox1` (внизу формы).

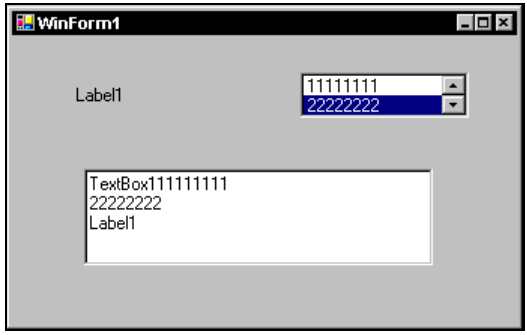


Рис. 4.6. Компоненты, реализующие механизм Drag and Drop

Ниже приведены обработчики событий `MouseDown` (для источников), а также `DragDrop` и `DragEnter` (для приемника), требуемые для реализации механизма Drag and Drop.

```
// обработчик события DragDrop для приемника
procedure TWinForm1.TextBox1_DragDrop(sender: System.Object; e:
System.Windows.Forms.DragEventArgs);
```

```
var
   S : String;
begin
  // можно предусмотреть очистку содержимого компонента TextBox1
  //TextBox1.Clear;
  S := string(e.Data.GetData(DataFormats.Text));
  // символы #13+#10 обеспечивают перевод строки и возврат каретки
  TextBox1.AppendText(S+#13+#10);
end;

// обработчик события DragEnter для приемника
procedure TWinForm1.TextBox1_DragEnter(sender: System.Object;
e: System.Windows.Forms.DragEventArgs);
begin
  if e.Data.GetDataPresent(DataFormats.Text) then
  e.Effect := DragDropEffects.Copy;
end;

// обработчик события MouseDown для источника
procedure TWinForm1.ListBox1_MouseDown(sender: System.Object;
e: System.Windows.Forms.MouseEventArgs);
var
SItems: System.Object;
dropEffect: DragDropEffects;
begin
if e.Button=System.Windows.Forms.MouseButtons.Left then
begin
   if ListBox1.SelectedIndex<>-1 then
   begin
     SItems:=ListBox1.Items[ListBox1.SelectedIndex];
     ListBox1.DoDragDrop(SItems,DragDropEffects.Copy);
   end;
end;
end;

// обработчик события MouseDown для еще одного источника
procedure TWinForm1.Label1_MouseDown(sender: System.Object;
e: System.Windows.Forms.MouseEventArgs);
var
s:System.Object;
begin
//dropEffect: DragDropEffects;
if e.Button=System.Windows.Forms.MouseButtons.Left then
begin
s:=Label1.Text;
Label1.DoDragDrop(s,DragDropEffects.Copy);
end;
end;
```

Метод `GetData` возвращает данные, связанные с определенным форматом. В примере строка `DataFormats.Text` означает предопределенное имя формата для текстовых данных.

Для всех компонентов, размещенных на форме и участвующих в перетаскивании методом Drag and Drop в качестве источника или приемника, нужно установить их свойству `AllowDrop` значение `True`.

Если с помощью рассматриваемого механизма выполняется не копирование, а перемещение объекта, то в обработчике события `MouseDown` для источника нужно предусмотреть его очистку от перемещенного объекта. Например, так:

```
procedure TWinForm1.Label1_MouseDown(sender: System.Object;
e: System.Windows.Forms.MouseEventArgs);
var
s:System.Object;
dropEffect: DragDropEffects;
begin
if e.Button=System.Windows.Forms.MouseButtons.Left then
begin
s:=Label1.Text;
 dropEffect:= Label1.DoDragDrop(s,dragdropEffects.Move);
  if dropEffect = dragdropEffects.Move then
   Label1.Text:='';
end;
end;
```

Ясно, что в нашем обработчике события DragEnter для приемника при этом должна быть задана строка:

```
e.Effect := DragDropEffects.Move;
```

ГЛАВА 5

Работа с текстом

В этой главе мы рассмотрим компоненты, используемые для отображения, редактирования и ввода текста в приложениях Windows Forms. Как отмечалось, соответствующие компоненты расположены на странице **Windows Forms** Палитры инструментов и принадлежат пространству имен `System.Windows.Forms`.

Отображение текста

Текст представляет собой надпись (ярлык) и чаще всего используется в качестве заголовков для других элементов управления. Обычно для отображения надписей используется компонент `Label`, называемый также *меткой*. Он представляет собой *простой текст*, который пользователь не может отредактировать при выполнении приложения. Программно этот текст можно изменять при выполнении приложения.

С помощью компонента `Label`, в отличие от его аналога в VCL, можно также задавать отображение графических изображений.

Для управления автоматической *коррекцией размеров* компонента `Label` в зависимости от текста надписи служит свойство `AutoSize` типа `Boolean`. Если свойство имеет значение `True`, то компонент `Label` изменяет свои размеры соответственно содержащемуся в нем тексту, заданному в свойстве `Text`.

Способ *выравнивания текста* внутри компонента `Label` задает свойство `TextAlign` типа `ContentAlignment`, которое может принимать одно из следующих значений:

- `TopLeft` (вверху слева, по умолчанию);
- `TopCenter` (вверху в центре);
- `TopRight` (вверху справа);
- `MiddleLeft` (посредине слева);

- `MiddleCenter` (посредине в центре);
- `MiddleRight` (посредине справа);
- `BottomLeft` (внизу слева);
- `BottomCenter` (внизу в центре);
- `BottomRight` (внизу справа).

Замечание

Выбранный с помощью свойства `TextAlign` вариант выравнивания (справа или слева) изменится на противоположный (слева или справа) в случае, если свойство `RightToLeft` имеет значение `Yes`.

С помощью свойства `Image` задается имя файла с изображением, которое отображается вместе с текстом.

Способ выравнивания изображения внутри компонента `Label` задает свойство `ImageAlign`, действующее так же, как и свойство `TextAlign`.

Отображаемое на компоненте `Label` изображение можно задавать с помощью свойств `ImageList` и `ImageIndex`. Для этого нужно создать компонент `ImageList` (например, с именем `ImageList1`) и поместить в него одно или несколько изображений. Свойству `ImageList` компонента `Label` следует выбрать в качестве значения имя этого компонента (`ImageList1`) и указать номер изображения с помощью свойства `ImageIndex`.

Текст с гиперссылкой

Текст (надпись) с гиперссылкой задается с помощью компонента `LinkLabel`. С его помощью можно задать одну или несколько гиперссылок. Большой необходимости в установке нескольких гиперссылок с помощью одного компонента `LinkLabel` обычно не возникает, поэтому данного вопроса мы касаться не будем.

Для обеспечения возможности использования компонента гиперссылки, нужно выполнить следующее. В качестве значения свойства `Text` следует поместить строку текста, часть которого или текст в целом будет представлять гиперссылку. По умолчанию гиперссылкой является весь текст, задаваемый с помощью свойства `Text`. Выделение части строки, играющей роль гиперссылки, можно выполнить с помощью редактора свойства `LinkArea`. Для этого достаточно щелчком на кнопке с тремя точками в строке свойства Инспектора объектов вызвать редактор этого свойства (рис. 5.1) и выделить нужную часть строки текста.

Свойство `LinkVisited` определяет необходимость автоматического отображения на тексте гиперссылки (путем изменения цвета гиперссылки) события ее выбора. При этом гиперссылка получает цвет ранее выбранной ссылки, но само событие выбора гиперссылки при этом не происходит.

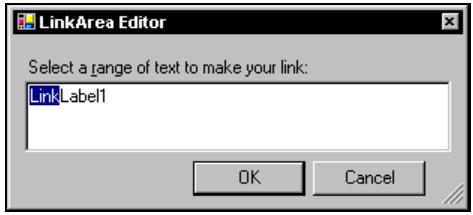


Рис. 5.1. Редактор свойства `LinkArea`

Цвет гиперссылки определяет свойство `LinkColor` типа `Color`. По умолчанию — ярко-синий цвет.

Цвет ранее выбранной гиперссылки определяет свойство `VisitedLinkColor` типа `Color`. По умолчанию — фиолетовый цвет.

Цвет гиперссылки в момент ее выбора определяет свойство `ActiveLinkColor` типа `Color`. По умолчанию — красный цвет.

Чтобы задать гиперссылку, в процедуру обработки события создания формы или в процедуру инициализации формы следует поместить строку вида:

```
Self.LinkLabel1.Links[0].LinkData:='www.microsoft.com';
```

Здесь устанавливается связь первой гиперссылки с адресуемым с ее помощью ресурсом Интернета.

Процедура обработки события выбора гиперссылки может содержать следующий код:

```
procedure TWinForm13.LinkLabel1_LinkClicked(sender: System.Object;
e: System.Windows.Forms.LinkLabelLinkClickedEventArgs);
var
target: String;
begin
   LinkLabel1.LinkVisited:=True;
   target:=string(e.Link.LinkData);
   System.Diagnostics.Process.Start(target);
end;
```

Здесь для перехода по гиперссылке к ресурсу Интернета используется метод `System.Diagnostics.Process.Start`. Для обеспечения возможности использования этого метода в предложении `Uses` модуля формы следует поместить строку `System.Diagnostics`.

Простейший текстовый редактор

Простейший текстовый редактор задается посредством компонента `TextBox`. С его помощью пользователь может вводить текст в приложении. Обычно этот

компонент служит для отображения, ввода или редактирования *одной* строки текста. При этом для ввода и отображения текста у рассматриваемого компонента используется его свойство `Text` типа `string`.

При необходимости можно установить *многострочный* режим ввода текста. Для чего следует свойству `Multiline` типа `Boolean` задать значение `True`. В этом случае текст вводится и отображается с помощью свойства `Lines` типа `array of string`. Соответственно доступ к строкам текста, хранимого в виде массива, осуществляется с помощью индексирования.

При многострочном режиме ввода и редактирования текста требуется управлять способом переноса слов. Делается это с помощью свойства `WordWrap` типа `Boolean`. При значении `True` свойства слова в области правого края поля автоматически переносятся на следующую строку (рис. 5.2). Если свойство имеет значение `False`, то текст вводится пользователем в одной строке с автоматическим прокручиванием влево вплоть до выполнения им переноса самостоятельно.

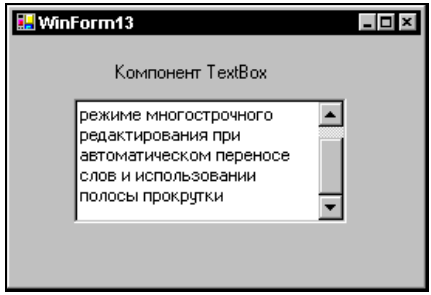


Рис. 5.2. Вид компонента в многострочном режиме

При использовании многострочного режима ввода и редактирования текста с помощью свойства `ScrollBars` типа `ScrollBars` можно управлять полосами прокрутки рассматриваемого компонента (рис. 5.2). Тип `ScrollBars` является перечислимым и содержит следующие значения: `None` (нет), `Horizontal` (горизонтальная), `Vertical` (вертикальная) и `Both` (обе).

В режиме многострочного ввода и редактирования текста свойство `AcceptsReturn` типа `Boolean` определяет, какой будет реакция на нажатие клавиши <Enter>. Если свойство имеет значение `True`, то при нажатии клавиши <Enter> произойдет создание очередной строки текста, при значении `False` (по умолчанию) нажатие клавиши активизирует кнопку по умолчанию на форме. В этом случае для создания новой строки текста пользователь должен нажать комбинацию клавиш <Ctrl>+<Enter>. Если кнопка по умолчанию на форме отсутствует, то нажатие клавиши <Enter> будет всегда приводить к созданию новой строки.

Свойство `Font` типа `Font` позволяет задавать с помощью Инспектора объектов параметры шрифта текста, отображаемого в окне компонента `TextBox`. Программно

составляющие этого свойства непосредственно менять нельзя. В частности, недопустимой является строка кода вида:

```
TextBox1.Font.Bold:=True;
```

Чтобы задать процедуру установки/отмены полужирного начертания текста компонента `TextBox1`, можно создать следующий обработчик события нажатия кнопки `Button5`:

```
procedure TWinForm13.Button5_Click(sender: System.Object;
e: System.EventArgs);
var
    NewFontStyle: FontStyle;
begin
    TextBox1.Focus;
  if TextBox1.Font.Bold then
    NewFontStyle := TextBox1.Font.Style xor FontStyle.Bold
  else
    NewFontStyle := TextBox1.Font.Style or FontStyle.Bold;
    TextBox1.Font := System.Drawing.Font.Create(TextBox1.Font.Name,
    TextBox1.Font.Size, NewFontStyle);
end;
```

Здесь для установки/отмены полужирного начертания текста в компоненте используется метод `Create`, принадлежащий пространству имен `System.Drawing`.

С помощью свойства `CharacterCasing` типа `CharacterCasing` можно задать ввод и редактирование текста в обычном режиме, с использованием верхнего или нижнего регистра. Тип `CharacterCasing` является перечислимым и содержит следующие значения: `Lower` (нижний), `Normal` (обычный) и `Upper` (верхний). При задании (программно или с помощью Инспектора объектов) варианта `Lower` или `Upper` весь набранный или вновь вводимый текст будет отображаться в нижнем или верхнем регистре соответственно. При задании варианта `Normal` регистр уже набранного текста не изменяется, а новый текст можно вводить, используя символы верхнего и нижнего регистра.

С помощью свойства `PasswordChar` типа `char` можно установить символ, который будет отображаться взамен каждого из символов текста. Тем самым можно обеспечить маскирование всех символов текста, которое обычно используется при вводе паролей. Например, в обработчике события нажатия кнопки `Button1` можно поместить следующий код:

```
procedure TWinForm13.Button1_Click(sender: System.Object;
e: System.EventArgs);
begin
  TextBox1.PasswordChar:='*';
  TextBox1.MaxLength:=10;
end;
```

Здесь в качестве маски для вводимого текста пароля использован символ '*', а с помощью свойства `MaxLength` установлено ограничение в 10 символов на длину вводимого пароля.

Посредством свойства `ReadOnly` типа `Boolean` можно программно запретить ввод и редактирование текста. Для этого названному свойству нужно задать значение `True`.

Редактор текста с форматированием

Редактор текста с форматированием представляет собой компонент `RichTextBox`. Этот компонент предоставляет больше функциональных возможностей по форматированию текста, чем простейший текстовый редактор `TextBox`.

Важными отличительными составляющими рассматриваемого компонента являются методы `LoadFile(path: string)` и `SaveFile(path: string)`, позволяющие загружать и сохранять данные в формате Rich Text Format (RTF) соответственно.

Свойство `Rtf` типа `string` содержит текст вместе с управляющими символами форматирования.

С помощью компонента можно форматировать отдельные фрагменты вводимого и редактируемого текста. Для этого соответствующий фрагмент текста нужно предварительно выделить. Выделение текста может быть выполнено пользователем с помощью мыши или программно. Для этого предназначен метод `Select(start: Integer; length: Integer);`. Он выделяет фрагмент длиной `length` символов, начиная от символа с номером `start`.

Задать параметры шрифта и цвета для выделенного фрагмента можно с помощью свойств `SelectionFont` типа `Font` и `SelectionColor` типа `Color` соответственно.

С помощью перегружаемого метода `Find` можно выполнить поиск фрагмента текста, искомый текст может быть задан с помощью строки типа `string` или массива элементов типа `char`. По результатам поиска метод возвращает порядковый номер первого символа найденного текста или `-1` в случае неудачного поиска.

Обмен данными с текстовыми процессорами

Способность компонента загружать и сохранять файлы в формате RTF фактически означает возможность его использования для обмена данными с текстовыми процессорами. К примеру, при работе с базами данных в приложении Delphi для .NET данные можно помещать в компонент `RichTextBox`, сохранять их в файле RTF и далее открывать для работы в текстовом процессоре, например Microsoft Word.

И наоборот, можно подготовить требуемые данные в текстовом процессоре, сохранить в файле RTF и затем загрузить этот файл с помощью компонента `RichTextFormat` для последующей обработки в приложении Delphi для .NET.

Рассмотрим пример использования компонента `RichTextFormat` для ввода, редактирования и форматирования текста, а также для загрузки и сохранения файлов RTF. Такое приложение может использоваться для обмена данными с текстовым процессором. Вид формы приложения приведен на рис. 5.3.

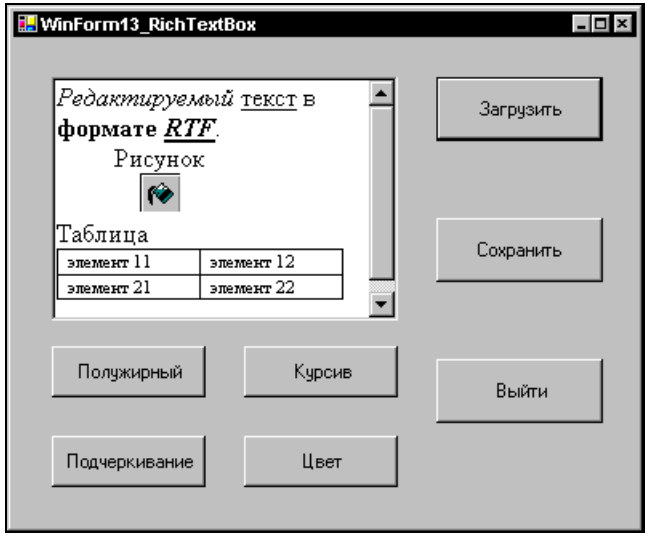


Рис. 5.3. Вид формы приложения с редактором `RichTextBox`

Как видим из содержимого приведенной формы приложения (рис. 5.3), в файле RTF и компоненте `RichTextBox`, кроме текста, могут находиться таблицы и рисунки.

Модуль формы для рассматриваемого примера (без ряда комментариев) содержит следующий код:

```
unit WinFormRichTextBox;
interface
uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data;
type
  TWinForm13 = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    Components: System.ComponentModel.Container;
    RichTextBox1: System.Windows.Forms.RichTextBox;
```

```
    Load: System.Windows.Forms.Button;
    Save: System.Windows.Forms.Button;
    Bold: System.Windows.Forms.Button;
    Italic: System.Windows.Forms.Button;
    Underline: System.Windows.Forms.Button;
    Color: System.Windows.Forms.Button;
    Closefm: System.Windows.Forms.Button;
    ColorDialog1: System.Windows.Forms.ColorDialog;
    procedure InitializeComponent;
    procedure Load_Click(sender: System.Object; e: System.EventArgs);
    procedure Save_Click(sender: System.Object; e: System.EventArgs);
    procedure Bold_Click(sender: System.Object; e: System.EventArgs);
    procedure Italic_Click(sender: System.Object; e: System.EventArgs);
    procedure Underline_Click(sender: System.Object; e: System.EventArgs);
    procedure Color_Click(sender: System.Object; e: System.EventArgs);
    procedure Closefm_Click(sender: System.Object; e: System.EventArgs);
  {$ENDREGION}
  strict protected
    procedure Dispose(Disposing: Boolean); override;
  private
    { Private Declarations }
  public
    constructor Create;
  end;

  [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm13))]

implementation

{$AUTOBOX ON}

{$REGION 'Windows Form Designer generated code'}

procedure TWinForm13.InitializeComponent;
begin
  Self.RichTextBox1 := System.Windows.Forms.RichTextBox.Create;
  Self.Load := System.Windows.Forms.Button.Create;
  Self.Save := System.Windows.Forms.Button.Create;
  Self.Bold := System.Windows.Forms.Button.Create;
  Self.Italic := System.Windows.Forms.Button.Create;
  Self.Underline := System.Windows.Forms.Button.Create;
  Self.Color := System.Windows.Forms.Button.Create;
  Self.Closefm := System.Windows.Forms.Button.Create;
  Self.ColorDialog1 := System.Windows.Forms.ColorDialog.Create;
  Self.SuspendLayout;
```

```
//
// RichTextBox1
//
Self.RichTextBox1.Location := System.Drawing.Point.Create(24, 24);
Self.RichTextBox1.Name := 'RichTextBox1';
Self.RichTextBox1.Size := System.Drawing.Size.Create(216, 152);
Self.RichTextBox1.TabIndex := 0;
Self.RichTextBox1.Text := 'RichTextBox1';
//
// Load
//
Self.Load.Location := System.Drawing.Point.Create(264, 24);
Self.Load.Name := 'Load';
Self.Load.Size := System.Drawing.Size.Create(104, 40);
Self.Load.TabIndex := 1;
Self.Load.Text := 'Загрузить';
Include(Self.Load.Click, Self.Load_Click);
//
// Save
//
Self.Save.Location := System.Drawing.Point.Create(264, 112);
Self.Save.Name := 'Save';
Self.Save.Size := System.Drawing.Size.Create(104, 40);
Self.Save.TabIndex := 2;
Self.Save.Text := 'Сохранить';
Include(Self.Save.Click, Self.Save_Click);
//
// Bold
//
Self.Bold.Location := System.Drawing.Point.Create(24, 192);
Self.Bold.Name := 'Bold';
Self.Bold.Size := System.Drawing.Size.Create(96, 32);
Self.Bold.TabIndex := 3;
Self.Bold.Text := 'Полужирный';
Include(Self.Bold.Click, Self.Bold_Click);
//
// Italic
//
Self.Italic.Location := System.Drawing.Point.Create(144, 192);
Self.Italic.Name := 'Italic';
Self.Italic.Size := System.Drawing.Size.Create(96, 32);
Self.Italic.TabIndex := 4;
Self.Italic.Text := 'Курсив';
Include(Self.Italic.Click, Self.Italic_Click);
//
```

```
  // Underline
  //
  Self.Underline.Location := System.Drawing.Point.Create(24, 248);
  Self.Underline.Name := 'Underline';
  Self.Underline.Size := System.Drawing.Size.Create(96, 32);
  Self.Underline.TabIndex := 5;
  Self.Underline.Text := 'Подчеркивание';
  Include(Self.Underline.Click, Self.Underline_Click);
  //
  // Color
  //
  Self.Color.Location := System.Drawing.Point.Create(144, 248);
  Self.Color.Name := 'Color';
  Self.Color.Size := System.Drawing.Size.Create(96, 32);
  Self.Color.TabIndex := 6;
  Self.Color.Text := 'Цвет';
  Include(Self.Color.Click, Self.Color_Click);
  //
  // Closefm
  //
  Self.Closefm.Location := System.Drawing.Point.Create(264, 200);
  Self.Closefm.Name := 'Closefm';
  Self.Closefm.Size := System.Drawing.Size.Create(104, 40);
  Self.Closefm.TabIndex := 7;
  Self.Closefm.Text := 'Выйти';
  Include(Self.Closefm.Click, Self.Closefm_Click);
  //
  // TWinForm13
  //
  Self.AutoScaleBaseSize := System.Drawing.Size.Create(5, 13);
  Self.ClientSize := System.Drawing.Size.Create(408, 383);
  Self.Controls.Add(Self.Closefm);
  Self.Controls.Add(Self.Color);
  Self.Controls.Add(Self.Underline);
  Self.Controls.Add(Self.Italic);
  Self.Controls.Add(Self.Bold);
  Self.Controls.Add(Self.Save);
  Self.Controls.Add(Self.Load);
  Self.Controls.Add(Self.RichTextBox1);
  Self.Name := 'TWinForm13';
  Self.Text := 'WinForm13_RichTextBox';
  Self.ResumeLayout(False);
end;
{$ENDREGION}
```

```
procedure TWinForm13.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
  inherited Dispose(Disposing);
end;

constructor TWinForm13.Create;
begin
  inherited Create;
  InitializeComponent;
  //
  // TODO: Add any constructor code after InitializeComponent call
  //
end;

procedure TWinForm13.Closefm_Click(sender: System.Object;
e: System.EventArgs);
begin
 Self.Close;
end;

procedure TWinForm13.Color_Click(sender: System.Object; e: System.EventArgs);
begin
 ColorDialog1.ShowDialog;
 RichTextBox1.SelectionColor:=ColorDialog1.Color;
end;

procedure TWinForm13.Underline_Click(sender: System.Object;
e: System.EventArgs);
  var
    NewFontStyle: FontStyle;
begin
    RichTextBox1.Focus;
  if RichTextBox1.SelectionFont.Underline then
    NewFontStyle := RichTextBox1.SelectionFont.Style xor FontStyle.Underline
  else
    NewFontStyle := RichTextBox1.SelectionFont.Style or FontStyle.Underline;
    RichTextBox1.SelectionFont :=
System.Drawing.Font.Create(RichTextBox1.SelectionFont.Name,
    RichTextBox1.SelectionFont.Size, NewFontStyle);
end;
```

```
procedure TWinForm13.Italic_Click(sender: System.Object; e: System.EventArgs);
var
    NewFontStyle: FontStyle;
begin
    RichTextBox1.Focus;
  if RichTextBox1.SelectionFont.Italic then
    NewFontStyle := RichTextBox1.SelectionFont.Style xor FontStyle.Italic
  else
    NewFontStyle := RichTextBox1.SelectionFont.Style or FontStyle.Italic;
    RichTextBox1.SelectionFont :=
System.Drawing.Font.Create(RichTextBox1.SelectionFont.Name,
    RichTextBox1.SelectionFont.Size, NewFontStyle);
end;

procedure TWinForm13.Bold_Click(sender: System.Object; e: System.EventArgs);
var
    NewFontStyle: FontStyle;
begin
    RichTextBox1.Focus;
  if RichTextBox1.SelectionFont.Bold then
    NewFontStyle := RichTextBox1.SelectionFont.Style xor FontStyle.Bold
  else
    NewFontStyle := RichTextBox1.SelectionFont.Style or FontStyle.Bold;
    RichTextBox1.SelectionFont :=
System.Drawing.Font.Create(RichTextBox1.SelectionFont.Name,
    RichTextBox1.SelectionFont.Size, NewFontStyle);
end;

procedure TWinForm13.Save_Click(sender: System.Object; e: System.EventArgs);
begin
    RichTextBox1.SaveFile('F:\Books\Delphi_NET\5\Формат RTF.rtf');
end;

procedure TWinForm13.Load_Click(sender: System.Object; e: System.EventArgs);
begin
    RichTextBox1.LoadFile('F:\Books\Delphi_NET\5\Формат RTF.rtf');
end;
end.
```

С помощью кнопок, снабженных надписями, могут выполняться соответствующие действия. Например, обработчик события нажатия кнопки `Bold` с надписью `Полужирный` обеспечивает задание или отмену (если такое начертание уже установлено) полужирного начертания для выделенного фрагмента текста. Нажатие кнопки `Load` с надписью `Загрузить` вызывает загрузку файла со спецификацией `'F:\Books\Delphi_NET\5\Формат RTF.rtf'`. При нажатии кнопки `Color` с надписью

Цвет осуществляется вызов стандартного диалога ColorDialog1, используемого для установки цвета выделенного фрагмента текста.

Общие свойства редакторов

Рассмотрим некоторые важные общие свойства компонентов редакторов TextBox и RichTextBox, которые наследуются ими от родительского класса TextBoxBase.

Компоненты-редакторы позволяют копировать, вырезать и вставлять фрагменты текста с использованием буфера обмена. Для этого предназначены методы Cut, Copy и Paste соответственно. Например, строка кода

```
RichTextBox1.Copy;
```

задает копирование выделенного в компоненте RichTextBox1 фрагмента текста в буфер обмена.

Свойство AutoSize типа Boolean определяет, изменяется ли автоматически высота элемента управления (редактора) в соответствии с размером шрифта, определяемого свойством Font, при изменении размера шрифта. По умолчанию свойство имеет значение True и такое изменение высоты компонента происходит.

При использовании многострочного режима ввода и редактирования текста с помощью свойства AcceptsTabs типа Boolean можно задать, какой будет реакция на нажатие клавиши <Tab>. Если свойство имеет значение False (по умолчанию), то нажатие клавиши <Tab> вызовет передачу фокуса другому элементу управления. Если же свойство имеет значение True, нажатие клавиши <Tab> вызовет табуляцию текста относительно текущего положения курсора.

Свойства CanUndo и CanRedo типа Boolean определяют соответственно, можно ли выполнять отмену выполненных действий или повторное выполнение отмененных действий с помощью методов Undo и Redo. Выполняемые пользователем действия запоминаются в буфере. С помощью вызова метода Undo осуществляется отмена последнего выполненного действия. Например,

```
if (RichTextBox1.CanUndo=True) then RichTextBox1.Undo;
```

Последовательно вызывая этот метод, можно осуществить отмену целого ряда последних действий. С помощью метода ClearUndo можно очистить буфер со списком выполненных пользователем действий, которые можно было отменить. Например, после выполнения следующего кода:

```
RichTextBox1.ClearUndo;
```

Вызов метода Undo не приведет к отмене выполненных действий. Аналогичные пояснения могут быть сделаны относительно использования метода Redo.

ГЛАВА 6

Кнопки, списки, переключатели

В этой главе мы рассмотрим работу с важными компонентами, используемыми при создании интерфейса приложения Windows Forms: списками, кнопками и переключателями. Напомним, что соответствующие компоненты расположены на страницах **Windows Forms** и **Components** Палитры инструментов и принадлежат пространству имен `System.Windows.Forms`.

Работа с кнопками

Кнопки являются элементами управления и служат для выдачи команд на выполнение определенных функциональных действий, поэтому часто их еще называют *командными кнопками*. В приложении Windows Forms *командная кнопка*, или просто *кнопка*, представлена компонентом `Button`, который является оконным элементом управления. На поверхности кнопки могут располагаться текст (свойство `Text`) и/или графическое изображение.

В качестве текста на поверхности кнопки обычно указывают ее назначение кнопки или описание действий, выполняемых при нажатии.

Графическое изображение на поверхности кнопки можно задать с помощью свойства `Image` или `ImageList`. В первом случае нужно в Инспекторе объектов в строке свойства `Image` нажать кнопку с тремя точками и в открывшемся диалоге выбрать файл с изображением.

Второй вариант реализуется путем добавления в Конструктор формы компонента `ImageList` и установления связи с ним с помощью свойства `ImageList`. Номер выбранного в компоненте `ImageList` графического изображения задается с помощью свойства `ImageIndex`.

Замечание

Если задать изображение любым из указанных способов, то после компиляции приложения изображение на поверхности кнопки может не появиться. Это связано с

тем, что в файл ресурсов приложения требуемые изменения не вносятся автоматически. В такой ситуации следует закрыть проект приложения, сохранив внесенные изменения, и после запуска или открытия другого проекта приложения повторно открыть создаваемый проект и запустить его на компиляцию и выполнение.

Основным для кнопки является событие `Click`, возникающее при ее нажатии. При этом кнопка принимает соответствующий вид, подтверждая происходящее действие визуально. Действия, выполняемые в обработчике события `Click`, происходят сразу после отпускания кнопки.

Кнопку можно нажать следующими способами: щелчком мыши; нажатием клавиш <Space>, <Enter> или <Esc>, а также с помощью комбинации клавиш, если она задана в свойстве `Text`.

На щелчок мыши, нажатие клавиши <Space> или <Enter> реагирует находящаяся в фокусе кнопка, заголовок которой выделен пунктирным прямоугольником (рис. 6.1). Две кнопки, связываемые со свойством `AcceptButton` или `CancelButton` формы, реагируют соответственно на нажатие клавиши <Enter> или <Esc>, даже не находясь в фокусе ввода.

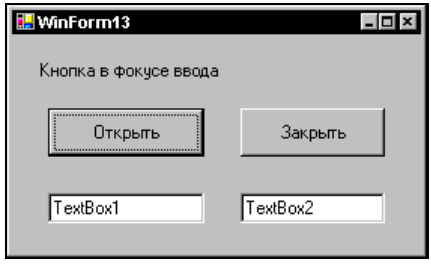


Рис. 6.1. Кнопка в фокусе ввода

Если фокус ввода получает не кнопочный элемент управления, например, `TextBox` или `Memo`, то может быть кнопка по умолчанию, которой становится так называемая кнопка "Accept" (принять), если она назначена. Кнопка "Accept" реагирует на нажатие клавиши <Enter> и в этом случае выделяется черным прямоугольником (рис. 6.2). Кнопки "Accept" обычно задают для кнопок, выполняющих подтверждение, например, **OK**. Чтобы произвольную кнопку задать как "Accept", необходимо свойству `AcceptButton` формы, на которой она расположена, установить в качестве значения имя этой кнопки. Сделать это можно с помощью Инспектора объектов или программно. По умолчанию свойство `AcceptButton` формы имеет значение `(none)`, и никакая кнопка не является кнопкой "Accept".

Программно установить кнопку "Accept", реагирующую на нажатие клавиши <Enter>, можно, например, так:

```
procedure TWinForm13.Button3_Click(sender: System.Object; e: System.EventArgs);
begin
  TWinForm13.ActiveForm.AcceptButton:=Button1;
end;
```

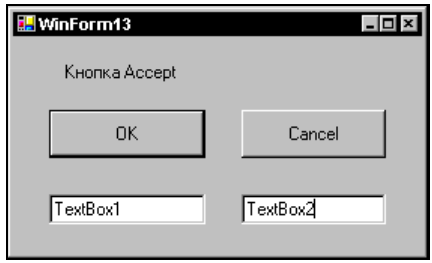


Рис. 6.2. Кнопка "Accept" не в фокусе ввода

Здесь в обработчике события нажатия кнопки `Button3` в качестве такой кнопки устанавливается кнопка `Button1`.

Событие `Click` может генерироваться для кнопки и в случае нажатия клавиши <Esc>, что обычно реализуется для кнопок, связанных с отменой какого-либо действия, например, кнопка **Cancel** в диалоговом окне. В приложениях Windows Forms соответствующие кнопки так и называют кнопка "Cancel" (отменить). Чтобы кнопка реагировала на нажатие клавиши <Esc> (задать ее как кнопка "Cancel"), необходимо свойству `CancelButton` формы, на которой расположена эта кнопка, задать в качестве значения имя этой кнопки. Сделать это можно с помощью Инспектора объектов или программно. По умолчанию свойство `CancelButton` формы имеет значение `(none)`, и никакая кнопка не реагирует на нажатие клавиши <Esc>.

Программно установить кнопку "Cancel", реагирующую на нажатие клавиши <Esc>, можно, например, так:

```
procedure TWinForm13.Button3_Click(sender: System.Object; e: System.EventArgs);
begin
  TWinForm13.ActiveForm.CancelButton:=Button2;
end;
```

Здесь в обработчике события нажатия кнопки `Button3` в качестве кнопки отмены устанавливается кнопка `Button2`.

Замечание

Если в фокусе ввода находится не кнопочный элемент управления, к примеру, многострочный редактор `TextBox`, то он первым получает и обрабатывает сообщение о нажатии клавиши <Enter> или <Esc>.

При отображении формы с помощью метода `ShowDialog` можно использовать свойство `DialogResult` кнопки `Button`, чтобы определить возвращаемое этим методом значение. Названное свойство можно устанавливать с помощью Инспектора объектов или программно, ему можно задавать следующие значения:

- `None` (по умолчанию);
- `OK`;
- `Cancel`;

- `Abort`;
- `Retry`;
- `Ignore`;
- `Yes`;
- `No`.

Внешний вид кнопки определяет свойство `FlatStyle`, оно может принимать следующие значения:

- `Flat` — кнопка имеет плоский вид;
- `Standard` — кнопка имеет объемный вид (по умолчанию);
- `Popup` — кнопка имеет плоский вид, при наведении на нее указателя мыши принимает объемный вид;
- `System` — внешний вид кнопки определяется операционной системой.

Цвет текста на кнопке определяет свойство `ForeColor` типа `Color`, а цвет фона, на котором отображается текст и или рисунок, — свойство `BackColor` типа `Color`.

Списки

Список представляет собой упорядоченную совокупность взаимосвязанных элементов, являющихся текстовыми строками или строками с изображениями. Списки применяются, например, для отображения информации о перечне шрифтов или о составе палитры цветов. Элементы списка могут быть отсортированы в алфавитном порядке или размещены в порядке добавления в список. Списки позволяют добавлять, удалять и выбирать отдельные их элементы (строки).

Простой список

Простой список представляет собой прямоугольную область, в которой располагаются его строковые элементы. Для работы с простым списком в приложении Windows Forms предназначен компонент `ListBox`. Для формирования простого списка при конструировании формы нужно в строке свойства `Items` щелчком на кнопке с тремя точками вызвать редактор коллекции строк (**String Collection Editor**) (рис. 6.3) и ввести в нем элементы списка.

Простой список можно также сформировать программно при выполнении приложения, используя для этого метод `Add` свойства `Items`. Например, добавление одного элемента в конец простого списка `ListBox1` выполняется с помощью строки вида:

```
ListBox1.Items.Add('Строка N');
```

Если количество строк больше, чем их может поместиться в видимой области списка, то у области отображения появляется полоса прокрутки. *Ориентация*

полосы прокрутки, а также *число столбцов*, которые одновременно видны в области списка, зависят от свойства `MultiColumn` типа `Boolean`. По умолчанию свойство имеет значение `False`. Это означает, что все элементы списка расположены в одном столбце, и при необходимости автоматически появляется (рис. 6.4, список справа) или исчезает (рис. 6.4, список слева) вертикальная полоса прокрутки.

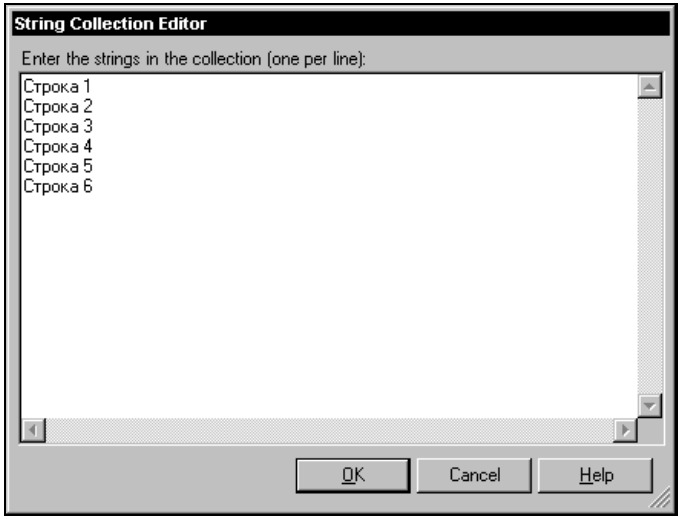


Рис. 6.3. Редактор коллекции строк

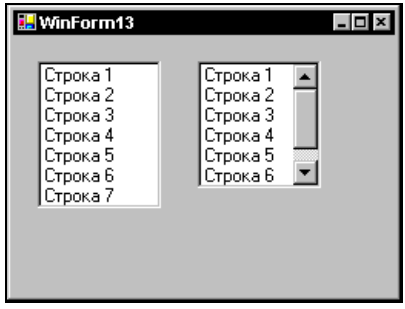


Рис. 6.4. Варианты списков

Если свойство `MultiColumn` имеет значение `True`, то в области списка может автоматически появляться горизонтальная полоса прокрутки, а элементы (в зависимости от высоты прямоугольной области) разбиваются на такое число столбцов, чтобы можно было, прокручивая список по горизонтали, просмотреть все его элементы. Число столбцов, которое отображается в видимой области списка, зависит от значения свойства `ColumnWidth`. Это свойство определяет ширину колонки в многоколоночном списке (свойство `MultiColumn` имеет значение `True`) в пик-

селах. По умолчанию свойство `ColumnWidth` имеет значение 0 и ширина колонки в области списка определяется исходной шириной колонки в редакторе **String Collection Editor** (рис. 6.5, список слева). Для сравнения на рис. 6.5 (список справа) приведен вид многоколоночного списка, у которого свойство `ColumnWidth` имеет значение `60`.

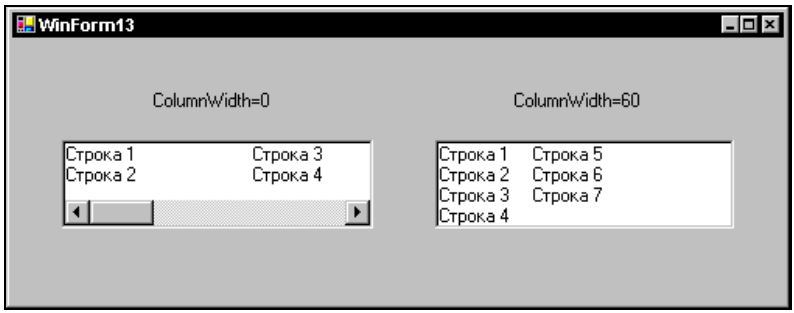


Рис. 6.5. Многоколоночные списки

Номер элемента, который выбран в простом списке, определяет свойство `SelectedIndex`. Если в списке нет выбранного элемента, то это свойство принимает значение `-1`.

При необходимости можно всегда отображать вертикальную полосу прокрутки. Для этого нужно свойству `ScrollAlwaysVisible` типа `Boolean` задать значение `True`.

В простом списке режим выбора элементов списка зависит от свойства `SelectMode`. Оно может принимать следующие значения: `None` — выбор элементов запрещен, `One` — можно выбрать только один элемент, `MultiSimple` — можно выбирать несколько элементов, `MultiExtended` — допускается расширенный выбор нескольких элементов: при нажатой клавише <Ctrl> можно выбрать произвольный набор элементов, при нажатой клавише <Shift> можно выбрать произвольный, но обязательно непрерывный, набор элементов.

При выборе в списке нескольких элементов значением свойства `SelectedIndices` является коллекция, которая содержит отсчитываемые с нуля индексы всех выбранных в текущий момент элементов простого списка `ListBox`. При этом значением свойства `SelectedIndices` является коллекция, которая содержит все выбранные в текущий момент элементы.

Комбинированный список

Комбинированный список объединяет поле редактирования и список. Пользователь может выбирать значение из списка или вводить его непосредственно в поле. В приложениях Windows Forms для работы с комбинированным списком служит компонент `ComboBox`.

В отличие от простого, комбинированный список не может иметь горизонтальную полосу прокрутки и допускает выбор только одного значения.

Список в компоненте `ComboBox` может быть постоянно раскрытым либо раскрывающимся. Свойство `DropDownStyle` типа `ComboBoxStyle` определяет *внешний вид* и *поведение* комбинированного списка. Оно принимает следующие значения (рис. 6.6):

- `DropDown` (раскрывающийся список с полем редактирования) — по умолчанию: пользователь может выбирать элементы в списке (при этом выбранный элемент появляется в поле ввода) или вводить (редактировать) информацию непосредственно в поле ввода;
- `Simple` (поле редактирования с постоянно раскрытым списком); чтобы список был виден полностью, необходимо увеличить высоту (подсвойство `Height` свойства `Size`) компонента `ComboBox`;
- `DropDownList` (раскрывающийся список, допускающий только выбор элементов в списке без возможности редактирования).

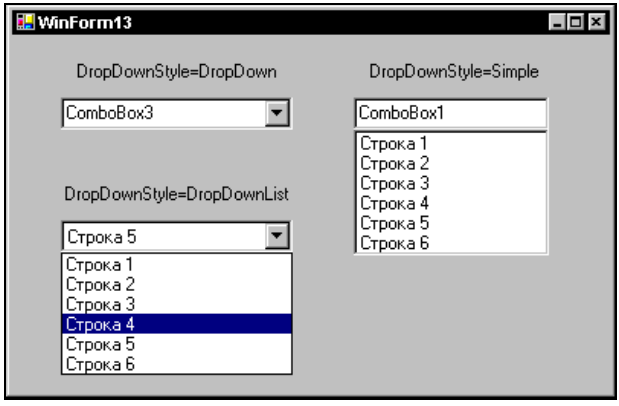


Рис. 6.6. Варианты комбинированного списка `ComboBox`

Свойство `MaxDropDownItems` типа `Integer` определяет максимальное *количество строк*, которые одновременно отображаются в раскрывающемся списке. Если значение этого свойства меньше числа строк списка, то у раскрывающегося списка автоматически появляется вертикальная полоса прокрутки. Если размер списка меньше, чем задано в свойстве `DropDownCount`, то отображаемая область списка автоматически уменьшается. Свойство `DropDownCount` по умолчанию имеет значение `8`.

Число строк определяет значение свойства `Items.Count`. Нумерация элементов списка ведется с нуля, последний элемент списка имеет номер `Count-1`. Номер выбранного элемента в списке определяет свойство `Items.SelectedIndex`. Если никакой элемент в списке не выбран, то это свойство имеет значение `-1`.

Рассмотрим пример использования комбинированного списка `ComboBox3` на форме, вид которой показан на рис. 6.7. С помощью кнопок, размещенных на форме, выполняется добавление нового элемента в комбинированный список из текстового поля (компонент `TextBox1`), удаление выбранного в комбинированном списке элемента или копирование выбранного в комбинированном списке элемента в текстовое поле (компонент `TextBox2`).

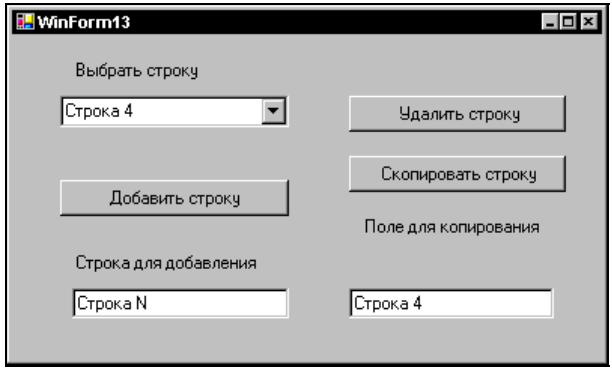


Рис. 6.7. Пример использования комбинированного списка

Обработчики событий нажатия кнопок, вызывающих указанные действия, содержат следующий код:

```
procedure TWinForm13.Button1_Click(sender: System.Object;
e: System.EventArgs);
begin
  //добавление нового элемента в комбинированный список
  ComboBox3.Items.Add(TextBox1.Text);
end;
procedure TWinForm13.Button2_Click(sender: System.Object;
e: System.EventArgs);
begin
  //копирование выбранного элемента в текстовое поле
  TextBox2.Text:=ComboBox3.GetItemText(ComboBox3.SelectedItem);
end;
procedure TWinForm13.Button3_Click(sender: System.Object;
e: System.EventArgs);
begin
  //удаление выбранного элемента из комбинированного списка
  ComboBox3.Items.RemoveAt(ComboBox3.SelectedIndex);
end;
```

В частности, нажатие кнопки `Button1` приводит к добавлению нового элемента в комбинированный список, а кнопки `Button2` — к копированию выбранного элемента в текстовое поле `TextBox2`.

Общие свойства списков

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

Элементы списка можно отсортировать в алфавитном порядке. Наличие или отсутствие *сортировки* определяет свойство `Sorted` типа `Bool`. При значении `False` этого свойства (по умолчанию) элементы в списке располагаются в порядке их поступления в список. Если же свойство `Sorted` имеет значение `True`, то элементы автоматически сортируются по алфавиту в порядке возрастания. На рис. 6.8 показаны не отсортированный (слева) и отсортированный (справа) списки. Если для отсортированного списка свойству `Sorted` снова установить значение `False`, то порядок элементов списка не изменится. В этом случае значение свойства сортировки будет действовать только для новых строк, добавляемых в список.

Рис. 6.8. Сортировка списков

Действие свойства `Sorted` является статическим, а не динамическим. Это означает, что при добавлении к отсортированному списку методами `Insert` и `Add` новых строк они размещаются на указанных позициях или в конце списка, а не по алфавиту. Чтобы отсортировать список, нужно сбросить значение свойства `Sorted` в `False`, а затем снова установить значение `True`:

```
ListBox1.Sorted  := False;
ListBox1.Sorted  := True;
```

Для кодирования символов, включающих русские буквы, применяется вариант Windows 1251 кода ANSI. В Windows сортировка этих символов осуществляется в порядке возрастания значений кодов с учетом регистра букв. Младшими по значению считаются специальные символы и разделители (такие как точка, тире, запятая и др.), затем следуют буквы латинского алфавита (рис. 6.8). При этом символы, отличающиеся только регистром, располагаются рядом: сначала — символ в нижнем регистре, затем — в верхнем регистре, несмотря на то, что их

коды не являются соседними. Последними упорядочиваются буквы русского алфавита, при этом символы, отличающиеся лишь регистром, также располагаются рядом.

Пользователь может *выбирать* отдельные строки списка с помощью мыши и клавиатуры. Выбранный в списке элемент определяется свойством `SelectedIndex` типа `Integer`. Отсчет строк начинается с нуля, поэтому, например, 2-я строка имеет номер 1. В приведенном на рис. 6.9 примере, чтобы отобразить номер выбранной в списке (`ListBox1`) строки, использован следующий обработчик события нажатия кнопки (`Button1`) с надписью `Указать № строки`:

```
procedure TWinForm13.Button1_Click(sender: System.Object; e: System.EventArgs);
begin
  TextBox1.Text:= 'В списке выбрана ' +
    IntToStr(ListBox1.SelectedIndex) + ' строка';
end;
```

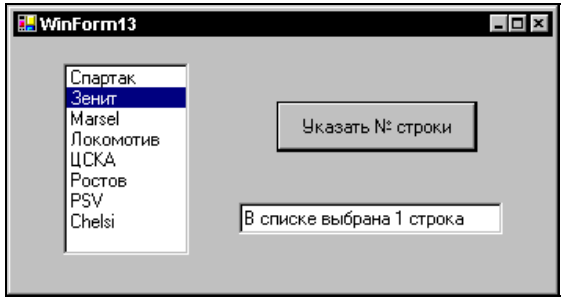


Рис. 6.9. Нумерация строк в списке

Как следует из приведенного кода обработчика, номер выбранной строки отображается в свойстве `Text` компонента `TextBox1`.

Замечание

Чтобы воспользоваться функцией `IntToStr`, нужно поместить код с названием пространства имен `Borland.Vcl.SysUtils` в предложение `Uses` модуля формы.

Программно можно выбрать элемент списка, установив свойству `SelectedIndex` требуемое значение. Так, инструкция `ListBox2.SelectedIndex:=3;` приводит к выбору четвертой строки списка `ListBox2`, и это отображается на экране.

Возникающие при выборе элемента списка события можно использовать для *обработки выбранных* строк. Например, щелчок левой кнопки мыши при выборе элемента списка можно обработать так:

```
procedure TWinForm13.ListBox1_MouseDown(sender: System.Object;
e: System.Windows.Forms.MouseEventArgs);
```

```
begin
 Label1.Text:= ListBox1.SelectedItem.ToString();
end;
```

Здесь надпись `Label1` отображает выбранный элемент списка `ListBox1`.

Переключатели

Переключатель (зависимый переключатель) позволяет выбрать единственное значение из определенного множества значений, представленного группой переключателей. Он может находиться в выбранном или невыбранном состоянии. Одновременно можно выбрать только один переключатель в группе; одиночный переключатель никогда не используется.

Флажок (независимый переключатель) отличается от переключателя тем, что в группе флажков одновременно можно установить флажки в любой комбинации (в том числе могут быть установлены или сброшены все флажки и т. д.). Флажок может находиться в установленном или сброшенном состоянии. Одиночный флажок часто используется, например, для включения/выключения какого-либо режима.

Анализ состояния переключателя или флажка позволяет выполнять соответствующие операции.

В приложении Windows Forms для работы с переключателями служат компоненты `CheckBox` и `RadioButton`. Иногда эти переключатели называют кнопками с фиксацией: `CheckBox` — с независимой фиксацией, а `RadioButton` — с зависимой.

Флажок — независимый переключатель

Флажок представлен компонентом `CheckBox`. Флажок действует независимо от других флажков, несмотря на то, что по функциональному назначению их часто объединяют в группы с общим названием, например, **Runtime Errors** (рис. 6.10).

Флажок выглядит как прямоугольник с текстовым заголовком. Если в нем есть галочка, то обозначенная этим флажком опция включена (или флажок *отмечен*). Если прямоугольник пуст, то флажок снят, или сброшен. Действия с одним флажком не отражаются на состоянии других флажков, если это не было специально предусмотрено.

В зависимости от значения свойства `ThreeState` типа `Boolean` флажок может находиться в двух или трех различных состояниях. Если свойство `ThreeState` имеет значение `False` (по умолчанию), то флажок может принимать два различных состояния: снятое и установленное. При этом для *определения состояния* флажка удобно использовать свойство `Checked` типа `Boolean`. По умолчанию оно имеет значение `False`, и флажок снят.

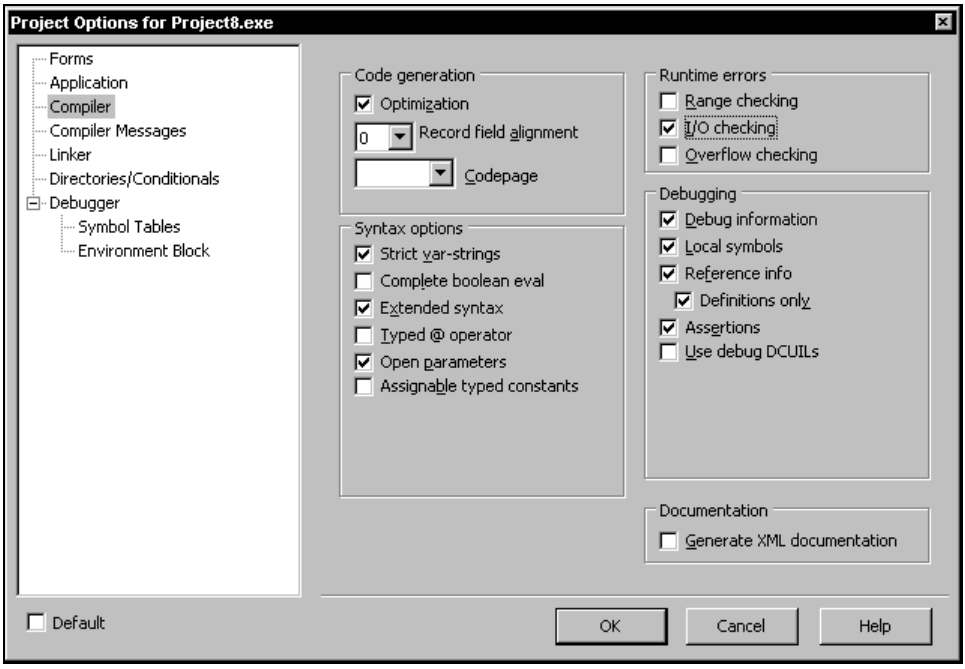


Рис. 6.10. Различные группы флажков

Пользователь может переключать состояние флажка щелчком мыши. Если флажок снят (не включен), то после щелчка он будет установлен (включен), и наоборот. При этом соответственно изменяется значение свойства `Checked`. Флажок можно переключить и с помощью клавиши <Space>, когда компонент `CheckBox` находится в фокусе ввода, а вокруг его заголовка отображен черный пунктирный прямоугольник.

Флажком можно управлять программно, устанавливая свойству `Checked` требуемые значения. Например:

```
CheckBox1.Checked:=True;
```

Если свойство `ThreeState` имеет значение `True`, то флажок может находиться в трех различных состояниях. При этом для *определения состояния* флажка используется свойство `CheckState` типа `CheckState`. Этот тип является перечислимым, и в соответствии с ним свойство `CheckState` может принимать следующие значения (рис. 6.11):

- `Checked` — флажок находится в установленном состоянии;
- `Indeterminate` — флажок находится в неопределенном состоянии. При этом состоянии флажок отображается с затенением;
- `UnChecked` — флажок находится в снятом состоянии.

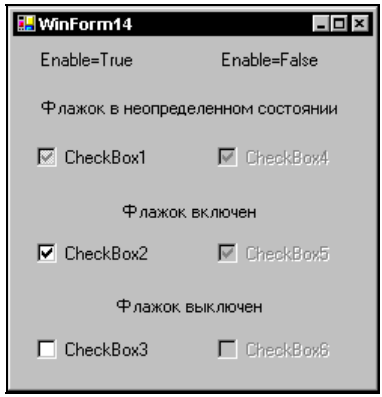


Рис. 6.11. Состояния флажка

Замечание

Свойства `Checked` и `CheckState` имеют взаимно-однозначное соответствие для установленного и сброшенного состояний флажка, изменение одного из них приводит к установке соответствующего значения другого.

Сделать флажок недоступным для изменения пользователем (заблокировать) можно установив свойству `Enabled` значение `False`:

```
CheckBox1.Enabled  := False;
```

После перехода флажка в заблокированный режим он сохраняет то состояние, в котором находился до выполнения блокировки. То есть неактивный флажок может находиться в установленном, снятом и неопределенном состояниях (рис. 6.11). При этом состояние флажка программно (с помощью свойства `Checked` или `CheckState`) можно изменять независимо от значения свойства `Enabled`.

Замечание

Галочка, отображаемая флажком в неопределенном состоянии (см. рис. 6.11), способна ввести в заблуждение, т. к. подобное состояние можно интерпретировать как включенное.

Свойство `Checked` имеет значение `True` только для выбранного режима флажка.

При изменении состояния флажка возникает событие `Click`, независимо от того, в какое состояние он переходит. В обработчике события `Click` обычно располагаются инструкции, проверяющие состояние флажка и выполняющие требуемые действия.

Приведем в качестве примера процедуру, производящую обработку события выбора состояния флажка.

```
procedure TWinForm14.CheckBox3_Click(sender: System.Object; e: System.EventArgs);
begin
  if (CheckBox3.CheckState=CheckState.Unchecked) then
   Label3.Text := 'Флажок выключен';
if (CheckBox3.CheckState=CheckState.Checked) then
   Label3.Text := 'Флажок включен';
if (CheckBox3.CheckState=CheckState.Indeterminate) then
   Label3.Text := 'Флажок в неопределенном состоянии';
end;
```

При переключении состояния флажка `CheckBox3` в надписи `Label3` отображается новое состояние этого флажка.

Зависимый переключатель

Переключатель представлен компонентом `RadioButton`. Соответствующие элементы управления отображаются в виде кружка с текстовым заголовком и/или рисунком. Переключатель может быть в выбранном или невыбранном состоянии (при выбранном состоянии в этом кружке появляется черная точка).

Переключатели обычно располагаются по группам, визуально выделенным в форме. Выбор переключателя является взаимоисключающим, т. е. при выборе одного переключателя другие становятся невыбранными. Каждый переключатель, помещенный в контейнер, автоматически включается в находящуюся на нем группу (рис. 6.12). Контейнерами обычно служат такие компоненты, как форма `Form`, панель `Panel` и группа `GroupBox`.

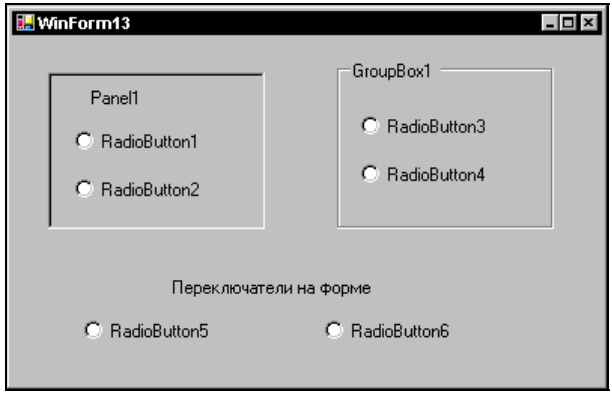


Рис. 6.12. Виды контейнеров для переключателей

При работе с группой один из переключателей рекомендуется делать выбранным по умолчанию, что можно выполнить при проектировании формы или в процессе выполнения приложения. Например, для приведенной на рис. 6.12 формы это можно выполнить так:

```
constructor TWinForm13.Create;
begin
  inherited Create;
  InitializeComponent;
  RadioButton1.Checked  := True;
  RadioButton3.Checked  := True;
  RadioButton5.Checked  := True;
end;
```

Когда в группе выбран один из переключателей, то его состояние нельзя изменить повторным щелчком. Отмена выбора переключателя происходит только при выборе другого переключателя из этой же группы.

Сделать отдельный переключатель недоступным для изменения пользователем (заблокировать) можно установив свойству `Enabled` значение `False`:

```
RadioButton1.Enabled  := False;
```

При этом состояние такого переключателя (если он находится в выбранном состоянии) может быть изменено пользователем путем выбора любого другого переключателя из той же группы. Кроме того, состояние заблокированного переключателя можно изменять программно (с помощью свойства `Checked`), независимо от значения свойства `Enabled`.

Установка изображения на переключателе выполняется с помощью свойства `Image` или `ImageList` так же, как и в случае с командной кнопкой `Button`.

Свойство `Dock` типа `DockStyle` задает местоположение компонента на контейнере, на котором он находится. Это свойство может принимать следующие значения:

- `None` — компонент располагается в зависимости от значения свойства `Location`;
- `Left` (`Right`) — компонент примыкает к левой (правой) границе контейнера;
- `Top` (`Buttom`) — компонент примыкает к верхней (нижней) границе контейнера;
- `Fill` — компонент занимает все доступное пространство контейнера.

Комбинация списка и флажка

Комбинацию списка `ListBox` и флажка `CheckBox` представляет компонент `CheckedListBox`. В этом компоненте флажок `CheckBox` отображается слева от каждого элемента списка. Компонент `CheckedListBox` содержит список элементов, по которому пользователь имеет возможность удобно перемещаться с помощью клавиатуры или полосы прокрутки. При этом пользователь может установить флажки слева от интересующих его элементов, это дает ему возможность перемещаться только по отмеченным элементам. Охарактеризуем кратко основные отличительные свойства рассматриваемого компонента.

Свойство `CheckedItems` типа `CheckedListBox.CheckedItemCollection` содержит коллекцию `CheckedListBox.CheckedItemCollection` отмеченных элементов списка.

Свойство `CheckedItems.Count` типа `Integer` возвращает число помеченных элементов списка.

Свойство `CheckedIndices` типа `CheckedListBox.CheckedIndexCollection` содержит коллекцию `CheckedListBox.CheckedIndexCollection` индексов отмеченных элементов списка.

Свойство `CheckOnClick` типа `Boolean` определяет способ переключения флажков для элементов списка. Если свойство имеет значение `False` (по умолчанию), то при первом щелчке мыши происходит выбор элемента списка, а при повторном щелчке — переключение состояния флажка. Если свойство имеет значение `True`, то переключение флажка происходит сразу при выборе элемента списка.

Для примера приведем вид формы приложения Windows Forms (рис. 6.13), в котором выполняется копирование отмеченных в компоненте `CheckedListBox1` элементов списка в список `ListBox1`.

Рис. 6.13. Форма для копирования выбранных элементов

Обработчик события `Click` нажатия кнопки `Button1`, вызывающей копирование отмеченных в компоненте `CheckedListBox1` элементов, содержит следующий код:

```
procedure TWinForm13.Button1_Click(sender: System.Object; e: System.EventArgs);
var i: integer;
begin
 ListBox1.Items.Clear();
 for i:=0 to CheckedListBox1.CheckedItems.Count-1  do
 ListBox1.Items.Add(CheckedListBox1.CheckedItems[i]);
end;
```

Здесь в обработчике выполняется очистка содержимого списка `ListBox1` и формирование нового состава этого списка путем добавления в него элементов с помощью свойства `CheckedListBox1.CheckedItems` рассматриваемого компонента.

ГЛАВА 7

Форма, контейнеры, приложение

В данной главе мы рассмотрим общую характеристику форм, приложений и контейнеров. При этом охарактеризуем общие свойства, методы и события формы, а также важные свойства и методы контейнеров и приложения.

Форма

Форма — это важнейший визуальный компонент приложения. Формы представляют собой видимые окна Windows и являются основной частью практически любого приложения Windows Forms. Термины "форма" и "окно" — синонимы, т. е. обозначают одно и то же.

Характеристика формы

В приложении Windows Forms форме соответствует класс `Form`. Этот класс может использоваться для создания стандартных и плавающих окон, панелей. С его помощью можно создавать модальные окна, такие как диалоги.

При разработке приложения Windows Forms создание формы для главного окна происходит автоматически вместе с созданием самого приложения. Напомним, что делается это по команде **File** | **New** | **Windows Forms Application – Delphi for NET** или путем выбора на Палитре инструментов варианта **Windows Forms Application**.

Для создания любой дополнительной формы нужно выполнить команду **File** | **New** | **Other** (Файл | Создать | Другой) и в открывшемся диалоговом окне **New Item** (Новый элемент) выбрать вариант **Windows Forms**. С этой же целью можно на Палитре инструментов в группе **Delphi for .NET Project** | **New Files** выбрать вариант **Windows Forms**.

Модули формы главного и дополнительного окон отличий не имеют. В файле проекта главное окно определяется в файле проекта с помощью атрибута

[STAThreade], который выделяет программе отдельный поток команд и запускает в нем модуль главного окна:

```
[STAThread]
begin
  Application.Run(TWinForm13.Create);
end.
```

Конструктор формы располагается в разделе implementation модуля формы и содержит следующий код:

```
constructor TWinForm13.Create;
begin
  inherited Create;
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent;
  //
  // TODO: Add any constructor code after InitializeComponent call
  //
end;
```

В процедуре InitializeComponent содержится код, который определяет внешний вид формы, а также состав и внешний вид компонентов, размещенных на форме. Например:

```
procedure TWinForm13.InitializeComponent;
begin
  Self.Button1 := System.Windows.Forms.Button.Create;
  Self.SuspendLayout;
  //
  // Button1
  //
  Self.Button1.DialogResult := System.Windows.Forms.DialogResult.Cancel;
  Self.Button1.Location := System.Drawing.Point.Create(96, 56);
  Self.Button1.Name := 'Button1';
  Self.Button1.Size := System.Drawing.Size.Create(80, 40);
  Self.Button1.TabIndex := 0;
  Self.Button1.Text := 'Button1';
  //
  // TWinForm13
  //
  Self.CancelButton:= Self.Button1;
  Self.AutoScaleBaseSize := System.Drawing.Size.Create(5, 13);
  Self.ClientSize := System.Drawing.Size.Create(292, 275);
```

```
  Self.Controls.Add(Self.Button1);
  Self.Name := 'TWinForm13';
  Self.Text := 'WinForm13';
  Self.ResumeLayout(False);
end;
```

Здесь содержится код, соответствующий форме с одной кнопкой. Форма имеет имя `TWinForm13`, в заголовке формы содержится текст `WinForm13`. Расположенная на форме кнопка установлена в качестве кнопки "Cancel", реагирующей на нажатие клавиши <Esc>.

В приложении Windows Forms файл проекта создает только одну форму — главную. Остальные формы нужно создавать динамически. Отметим, что главная форма не содержит автоматически создаваемых объявлений переменных типа формы для других форм. В связи с этим, для того чтобы из модуля одной формы, например `WinForm13`, можно было вызвать другую форму, например `TWinForm14`, нужно выполнить следующее:

1. В предложение `uses` модуля формы `WinForm13` включить имя модуля `WinForm14`.
2. Объявить переменную типа `TWinForm14`:

```
var
  WinForm14:TWinForm14;
```

Теперь можно выполнять создание экземпляра и вызов формы, например, как показано в обработчике:

```
procedure TWinForm13.Button2_Click(sender: System.Object; e: System.EventArgs);
begin
  WinForm14:=TWinForm14.Create;
  WinForm14.Show;
end;
```

Здесь по нажатию кнопки `Button2` на форме `WinForm13` создается экземпляр формы `TWinForm14`, и форма вызывается с помощью метода `Show`.

Свойства формы

Класс `Form` (рис. 7.1) является наследником класса `Control`, принадлежащего пространству имен `System.Windows.Forms` и непосредственно наследуемого от класса `Component`.

В *главе 4* (табл. 4.1) приведены некоторые важные свойства визуальных компонентов, унаследованные от класса `Control`. Класс `Form` имеет много других свойств, некоторые из них приведены в табл. 7.1.

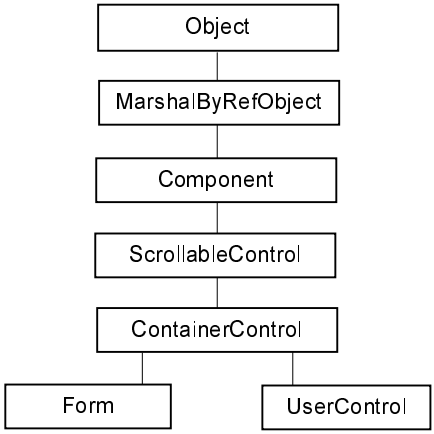


Рис. 7.1. Место класса `Form` в пространстве имен `System.Windows.Forms`

Таблица 7.1. *Некоторые свойства класса* `Form`

Свойство	Назначение
`ActiveForm: Form`	Получает активную форму приложения
`AutoScale: Boolean`	Получает или устанавливает автоматическое изменение размеров управляющих компонентов окна согласно размерам текущего шрифта
`AutoScroll: Boolean`	Получает или устанавливает значение, указывающее, поддерживает ли форма автоматическую прокрутку
`CancelButton: IButtonControl`	Получает или устанавливает кнопку "Cancel", реагирующую на нажатие клавиши <Esc>
`ClientSize: Size`	Получает или устанавливает размер клиентской области формы
`Icon: Icon`	Получает или устанавливает значок, связанный с формой
`IsMdiContainer: Boolean`	Получает или устанавливает значение, указывающее, является ли форма контейнером дочерних форм приложения MDI
`Menu: MainMenu`	Получает или устанавливает главное меню формы
`MdiChildren: array of Fom`	Получает массив дочерних форм, наследуемых от этой формы, для приложения MDI
`MdiParent: Form`	Получает или устанавливает для формы приложения MDI ее родительскую форму
`OwnedForms: array of Fom`	Получает массив подчиненных форм

Таблица 7.1 *(окончание)*

Свойство	Назначение
`Owner: Fom`	Получает или устанавливает владельца формы
`Parent: Fom`	Получает родителя формы
`Size: Size`	Получает или устанавливает размер формы
`WindowState: FomWindowState`	Получает или устанавливает состояние окна формы (обычно максимизировано или минимизировано)

Связи между формами

Дадим краткое пояснение относительно способов связи нескольких форм приложения. Форма может принадлежать другой форме — владельцу. Кроме того, для приложений с многодокументным интерфейсом (Multi Document Interface, MDI) форма может быть дочерней по отношению к родительской форме. Главная форма приложения не имеет родителей, но может иметь произвольное число дочерних форм.

Если некоторая форма *принадлежит* другой форме, то она минимизируется и закрывается вместе с формой владельцем. Например, если формой `Form2` владеет форма `Form1`, то при закрытии или минимизации формы `Form1` происходит закрытие или минимизация формы `Form2`. Принадлежащие формы никогда не отображаются сзади своей формы владельца. В качестве наглядного примера принадлежащей формы можно указать диалоговое окно **Найти и заменить** (Find and

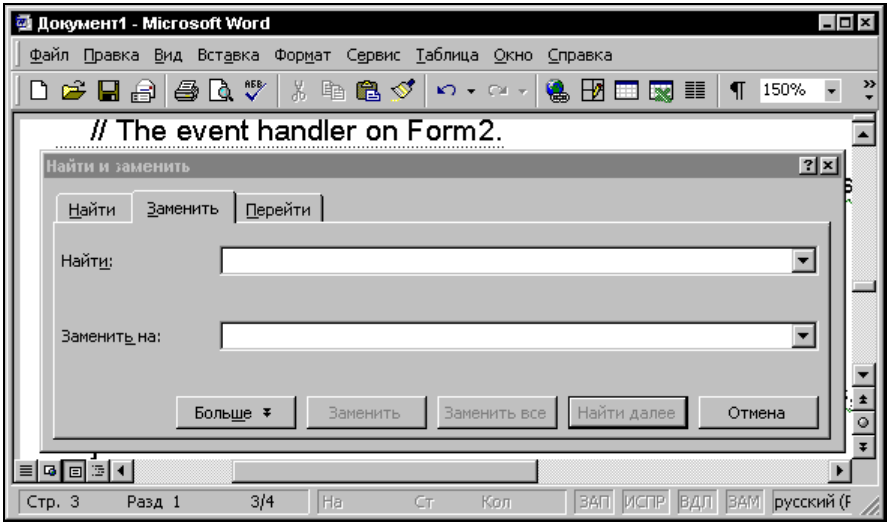


Рис. 7.2. Пример принадлежащей формы и формы владельца

Replace) (рис. 7.2), владельцем его является главное окно текстового процессора Microsoft Word.

Установить принадлежность одной формы другой можно с помощью метода `AddOwnedForm`. После назначения форма остается владельцем другой формы до вызова метода `RemoveOwnedForm`. Форму можно также установить как принадлежащую другой форме путем задания свойству `Owned` в качестве значения ссылки на форму владельца. Например:

```
procedure TWinForm13.Button1_Click(sender: System.Object; e: System.EventArgs);
begin
Self.AddOwnedForm(WinForm14);
end;
```

Здесь в обработчике события `Button1_Click` текущая форма `WinForm13` назначается в качестве владельца формы `WinForm14`.

Формы приложения MDI

Создаваемое приложение может иметь однодокументный (Single Document Interface, SDI) или многодокументный (MDI) интерфейс. В случае приложения с интерфейсом SDI информация отображается в одной форме. Если приложение имеет интерфейс MDI, то главная (родительская) форма может иметь несколько дочерних форм.

В качестве характерного примера приложения с интерфейсом MDI можно назвать табличный процессор Microsoft Excel. Укажем некоторые особенности дочерних форм:

- дочерняя форма ограничивается родительской формой, она всегда размещается внутри нее;
- дочерняя форма всегда отображается поверх родительской формы;
- дочерняя форма перемещается, максимизируется, минимизируется и закрывается вместе с родительской формой.

Для создания приложения MDI достаточно установить свойству `IsMdiContainer` главной формы значение `True`. Создаваемым дочерним формам нужно в качестве значения свойства `MdiParent` указать ссылку на родительскую форму.

Методы и события формы

В *главе 4* (табл. 4.2) приведены некоторые важные методы визуальных компонентов, унаследованные от класса `Control`. Напомним, что класс `Form` является наследником класса `Control`. Класс `Form` имеет много других методов, некоторые из них приведены в табл. 7.2.

Таблица 7.2. *Некоторые методы класса* `Form`

Метод	Назначение
`procedure Activate;`	Активизирует форму и передает ей фокус
`procedure AddOwnedForm(ownedForm: Form);`	Добавляет подчиненную форму к этой форме
`procedure Close;`	Закрывает форму
`procedure Dispose;`	Освобождает ресурсы, используемые компонентом
`procedure LayoutMdi(value: MdiLayout);`	Располагает дочерние формы приложения MDI внутри родительской формы
`procedure RemoveOwnedForm(ownedForm: Form);`	Удаляет подчиненную форму из этой формы
`procedure Show;`	Отображает форму
`procedure ShowDialog: DialogResult;`	Отображает форму как модальное диалоговое окно, не имеющее владельца
`procedure ShowDialog(owner: IWin32Window): DialogResult;`	Отображает форму как модальное диалоговое окно, имеющее владельца

В приложении Windows используются два вида диалогов (диалоговых окон): модальные и немодальные. *Модальные* диалоговые окна не дают возможности пользователю взаимодействовать с другими формами приложения, пока не завершится работа с ними. Примерами таких диалогов являются окна **About** (О программе) или **File Open** (Открыть файл). Для отображения формы как модального диалогового окна используется метод `ShowDialog`. Модальное диалоговое окно может иметь или не иметь владельца, в зависимости от этого вызывается соответствующий вариант метода `ShowDialog` (см. табл. 7.2).

Тип `DialogResult` результата вызова диалога является перечислимым и содержит следующие значения:

- `Abort` — нажата кнопка **Abort**;
- `Cancel` — нажата кнопка **Cancel**;
- `Ignore` — нажата кнопка **Ignore**;
- `No` — нажата кнопка **No**;
- `None` — ничего не возвращается, диалог остается активен;
- `OK` — нажата кнопка **OK**;
- `Retry` — нажата кнопка **Retry**;
- `Yes` — нажата кнопка **Yes**.

Обычно диалоговое окно должно содержать кнопки, с помощью которых пользователь может выбрать подходящий вариант закрытия диалога. Например, всегда должна быть кнопка **OK**. При необходимости выхода из диалога без принятия внесенных изменений нужно предусмотреть кнопку **Cancel**. Например, чтобы задать вариант закрытия диалога с помощью кнопки **Yes**, можно для события нажатия этой кнопки установить следующий обработчик:

```
procedure TWinForm14.Yes_Click(sender: System.Object; e: System.EventArgs);
begin
  self.DialogResult:=System.Windows.Forms.DialogResult.Yes ;
end;
```

При этом для формы-владельца, вызывающей модальный диалог `TWinForm14`, можно предусмотреть следующий код:

```
procedure TWinForm13.Button2_Click(sender: System.Object; e: System.EventArgs);
var
res: System.Windows.Forms.DialogResult;
begin
  WinForm14:=TWinForm14.Create;
  res:=WinForm14.ShowDialog(self);
  if res= System.Windows.Forms.DialogResult.Yes then Label1.Text:='Yes';
end;
```

Здесь, если результатом вызова модального окна является значение `Yes`, то свойство `Text` компонента `Label1` формы владельца получает значение `'Yes'`.

Немодальные диалоговые окна существуют рядом с основной формой, позволяя выполнять переключение между формой и диалогом. В частности, все дочерние формы являются немодальными диалоговыми окнами. В качестве примера немодального диалога можно указать диалоговое окно **Найти и заменить** (Find and Replace) (рис. 7.2). Для отображения формы как немодального диалогового окна нужно использовать метод `Show`.

В *главе 4* (табл. 4.3) приведен ряд важных *событий* визуальных компонентов, унаследованных от класса `Control`. Ряд других событий класса `Form` приведен в табл. 7.3.

Таблица 7.3. *Некоторые события класса* `Form`

Событие	Условие возникновения
`Activated: EventHandler;`	Активизация окна
`Closed: EventHandler;`	Форма закрыта
`Closing: CancelEventHandler;`	Форма закрывается
`Deactivated: EventHandler;`	Форма теряет фокус ввода
`Load: EventHandler;`	Перед тем как форма отображается первый раз

Таблица 7.3 *(окончание)*

Событие	Условие возникновения
`MdiChildActivate: EventHandler;`	Активизация или закрытие дочерней формы в приложении MDI
`MenuComplete: EventHandler;`	Меню формы теряет фокус
`MenuStart: EventHandler;`	Меню формы получает фокус

Контейнеры

При разработке приложения часто требуется выполнить объединение, или группирование, различных элементов управления. Группирование может понадобиться, например, при работе с переключателями на форме *(см. главу 6)* или при создании панели инструментов.

Объединение элементов выполняется с помощью специальных компонентов — контейнеров. *Контейнер* представляет собой визуальный компонент, на поверхности которого можно размещать другие компоненты; контейнер объединяет эти компоненты в группу и становится их владельцем. Владелец также отвечает за прорисовку своих дочерних элементов.

Как отмечалось в *главе 6*, для организации группы переключателей в приложениях Windows Forms контейнерами обычно служат такие компоненты, как форма `Form`, панель `Panel` и группа `GroupBox`.

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

Компоненты-контейнеры панель `Panel` и группа `GroupBox` являются визуальными и принадлежат пространству имен `System.Windows.Forms`. Многие свойства этих компонентов унаследованы от класса `Control` и приведены нами в *главе 4*. Укажем свойства и методы этих компонентов, характеризующие их как контейнеры.

При размещении на контейнере (панели `Panel` или группе `GroupBox`) каких-либо элементов управления все они наследуют свойства того контейнера, на котором они располагаются. Например, изменяя шрифт, цвет текста, цвет фона, видимость и доступность контейнера, мы одновременно автоматически изменяем свойства расположенных на нем элементов управления.

Свойство `Visible` типа `Boolean` получает или устанавливает значение, указывающее, отображается ли контейнер вместе со всеми размещенными на нем элементами управления. Например, с помощью следующей строки кода:

```
GroupBox1.Visible:=not GroupBox1.Visible;
```

можно изменять на противоположное состояние видимость группы `GroupBox1` вместе со всеми ее элементами управления.

Свойство `Enabled` типа `Boolean` получает или устанавливает значение, указывающее, доступен ли для взаимодействия с пользователем контейнер вместе со всеми размещенными на нем элементами управления. Например, с помощью следующей строки кода:

```
GroupBox1.Enabled:=CheckBox1.Checked;
```

можно устанавливать доступность группы `GroupBox1` вместе со всеми ее элементами управления в зависимости от состояния флажка `CheckBox1`. Если флажок установлен, то группа `GroupBox1` будет доступна, если флажок сброшен, то группа будет недоступна.

Замечание

Такие свойства, как шрифт, цвет текста, цвет фона и т. п., наследуются расположенными на контейнере элементами управления только в том случае, если они не подвергались индивидуальной настройке со стороны пользователя.

Компоненты панель `Panel` и группа `GroupBox` в основном схожи, но между ними можно указать три различия:

- для компонента `Panel` можно задавать стиль обрамления, а для компонента `GroupBox` — нельзя;
- компонент `Panel` не имеет строки заголовка, а компонент `GroupBox` может иметь;
- в отличие от `GroupBox`, компонент `Panel` может иметь полосу прокрутки.

Добавление элементов управления на контейнер (`GroupBox` или `Panel`) можно выполнить с помощью Конструктора формы, помещая их внутри границ компонента контейнера. Программно добавить к контейнеру элементы управления можно с помощью методов `Procedure Add(value: Control)` и `Procedure Remove(value: Control)`. Например:

```
Self.Button8 := System.Windows.Forms.Button.Create;
Self.Button8.Location := System.Drawing.Point.Create(72, 80);
Self.Button8.Name := 'Button8';
Self.Button8.Size := System.Drawing.Size.Create(160, 40);
Self.Button8.TabIndex := 8;
Self.Button8.Text := 'Кнопка 8';
Self.Panel1.Controls.Add(Self.Button8);
if(panel1.Controls.Contains(RadioButton1))
   then
     panel1.Controls.Remove(RadioButton1);
```

Здесь показано создание и добавление на компонент `Panel1` (панель) кнопки `Button8` и удаление с панели элемента управления `RadioButton1`, если он там имеется.

Свойство `HasChildren` типа `Boolean` компонента получает величину, указывающую, содержит ли компонент дочерние элементы управления. Если представ-

ляющее коллекцию свойство `Controls` имеет значение свойства `Count` большее нуля, то свойство `HasChildren` возвращает значение `True`.

Группа

Группа используется в основном для визуального выделения функционально связанных управляющих элементов. В приложениях Windows Forms для работы с группой служит компонент `GroupBox`, задающий прямоугольную рамку с необязательным заголовком (свойство `Text`) в верхнем левом углу и объединяющий содержащиеся в нем элементы управления. Например, на рис. 6.12 группа с заголовком `GroupBox1` используется для объединения зависимых переключателей `RadioButton3 и RadioButton4`.

Компонент `GroupBox` чаще всего используют для логического объединения нескольких зависимых переключателей (компонентов `RadioButton`), расположенных на форме. При этом в каждой группе в некоторый момент времени выбрать можно только один из зависимых переключателей.

Панель

Панель представляет собой контейнер, в котором можно размещать другие элементы управления. Панели применяются для группирования элементов управления, а также для создания панелей инструментов и строк состояния.

У панели свойство `BorderStyle` типа `BorderStyle` указывает стиль ее обрамления. Это свойство может принимать следующие значения:

- `None` — нет обрамления;
- `FixedSingle` — панель обрамлена одиночной линией;
- `Fixed3D` — панель имеет объемное обрамление.

Свойство `AutoScroll` типа `Boolean` получает или устанавливает значение, указывающее, может ли панель иметь автоматически устанавливаемые горизонталь-

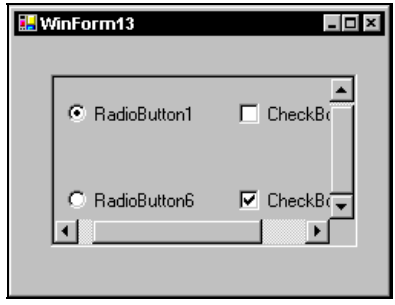


Рис. 7.3. Вид панели с полосами прокрутки

ную и/или вертикальную полосы прокрутки. На рис. 7.3 приведен вид панели с тремя элементами управления и двумя полосами прокрутки.

Показанная на рис. 7.3 панель имеет объемное обрамление.

Приложение

Само приложение Windows Forms представляется классом `Application`, принадлежащим пространству имен `System.Windows.Forms`. Этот класс имеет свойства и методы, позволяющие управлять приложением, например, методы для запуска и остановки приложения, для обработки сообщений Windows, а также свойства для получения информации о состоянии приложения. Объект класса `Application` создается автоматически для каждого приложения Windows Forms.

Напомним, что код запуска приложения автоматически помещается в файле проекта и имеет вид:

```
[STAThread]
begin
  Application.Run(TWinForm13.Create);
end.
```

Здесь выполняется запуск приложения с помощью метода `Run`.

Для примера приведем некоторые важные, на наш взгляд, свойства класса `Application`.

- `AllowQuit: Boolean` — получает значение, указывающее, может ли вызвавшая приложение программа выйти из него. В частности, это свойство возвращает значение `False`, если приложение было вызвано с помощью элемента управления, размещенного в Web-браузере. Это означает, что такой элемент управления не может выйти из приложения.
- `ExecutablePath: String` — получает путь доступа и имя исполняемого файла приложения. Например:

  ```
  procedure TWinForm13.Button1_Click(sender: System.Object;
  e: System.EventArgs);
  begin
  Label1.Text:=Application.ExecutablePath;
  end;
  ```

  Здесь в обработчике события нажатия кнопки `Button1` обеспечивается отображение на надписи `Label1` полного пути и имени исполняемого файла приложения.
- `StartupPath: String` — получает путь доступа к исполняемому файлу приложения без имени этого файла.

- `UserAppDataPath: String` — получает путь к пользовательским данным приложения.

Приведем также некоторые важные, на наш взгляд, методы класса `Application`.

- `procedure AddMessageFilter(value: IMessageFilter);` — добавляет фильтр для обработки сообщений Windows.
- `procedure DoEvents;` — обрабатывает все сообщения Windows из очереди сообщений. Отметим, что каждый раз, когда форма приложения реагирует на возникновение некоторого события, она осуществляет обработку всего кода, касающегося этого события. При этом приложение не реагирует на возникновение других событий, они находятся в очереди сообщений. В такой ситуации вызов метода `DoEvents` позволяет приостановить текущую обработку события (например, помещение данных в надпись) и выполнить обработку всех сообщений из очереди, например, для визуализации процесса перетаскивания данных.
- `procedure Exit;` — останавливает все циклы обработки сообщений по всем потокам и закрывает все окна приложения. Этот метод используется для закрытия приложения.
- `procedure RemoveMessageFilter;` — удаляет установленные фильтры для обработки сообщений Windows.
- `procedure Run;` — запускает стандартный цикл обработки сообщений текущего потока.

ГЛАВА 8

Диалоги, панель инструментов, меню

В этой главе дадим общую характеристику стандартных диалогов, панели инструментов и меню. При этом укажем основные свойства, методы и события соответствующих компонентов и опишем технику их создания. Отметим, что эти компоненты играют важную роль в организации пользовательского интерфейса приложения.

Диалоги

Рассмотрим стандартные диалоги, или диалоговые окна, которые используются для обеспечения интерфейса пользователя в случаях, когда пользователю приходится открывать или сохранять файлы, печатать документы, выбирать параметры печати, шрифта и цвет текста. Как отмечалось в *главе 4*, соответствующие им компоненты размещены на странице **Dialogs** Палитры инструментов и принадлежат пространству имен `System.Windows.Forms`.

При работе с любым стандартным диалогом соответствующий ему компонент, например `OpenFileDialog`, при помещении его в Конструктор формы размещается под формой, как показано на рис. 8.1.

Для *открытия* стандартных диалогов используется их метод `ShowDialog`. Этот метод возвращает результат типа `DialogResult`, который является перечислением и содержит следующие идентификаторы: **Abort**, **Cancel**, **Ignore**, **No**, **None**, **OK**, **Retry**, **Yes**. В общем случае эти идентификаторы (за исключением **None**) указывают на то, что диалоговое окно (не обязательно стандартное) в качестве результата возвращает одноименное значение. Например, идентификатор **Abort** указывает, что метод возвращает значение `Abort`. Обычно эти значения возвращаются при нажатии в диалоговом окне кнопки с одноименным названием. Идентификатор **None** указывает на возврат значения `Nothing`, означающего, что модальное диалоговое окно продолжает работать.

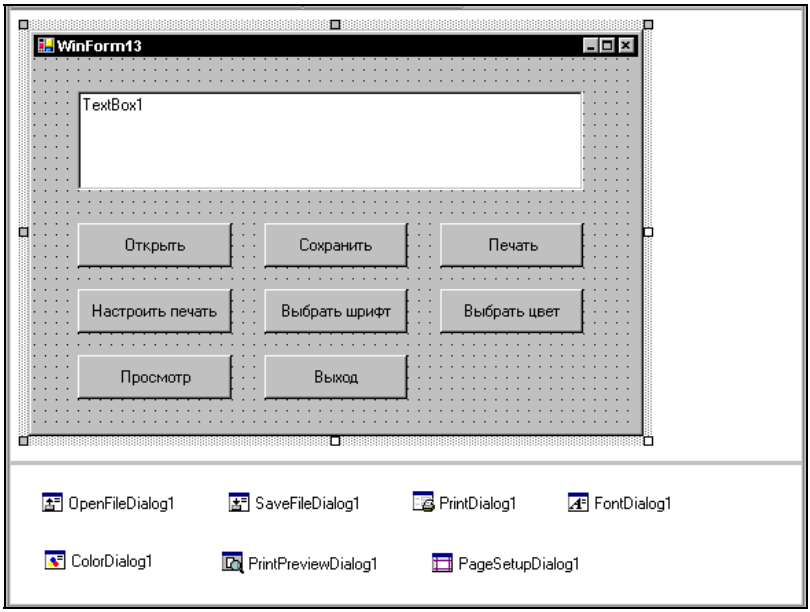


Рис. 8.1. Вид Конструктора формы приложения с диалогами

Для стандартных диалогов результатом завершения метода `ShowDialog` могут быть значения **Cancel** или **OK** в зависимости от того, какая кнопка была нажата пользователем **Cancel** (Отменить) или **OK**.

Диалоги открытия и закрытия файлов

В приложениях Windows Forms стандартные диалоги открытия и закрытия файлов задаются с помощью компонентов `OpenFileDialog` и `SaveFileDialog` соответственно. Эти компоненты имеют много общих свойств и методов. Вид стандартного диалога открытия файла документа показан на рис. 8.2. Стандартный диалог сохранения файла документа имеет во многом аналогичный вид.

Некоторые общие и важные, на наш взгляд, свойства диалогов открытия и закрытия файлов приведены в табл. 8.1.

Таблица 8.1. *Некоторые свойства диалогов открытия и закрытия файлов*

Свойство	Назначение
`AddExtension: Boolean`	Получает или устанавливает значение, указывающее, нужно ли автоматически добавлять расширение имени файла, если оно не было задано пользователем
`CheckFileExists: Boolean`	Получает или устанавливает значение, указывающее, нужно ли выдавать предупреждающее сообщение пользователю (`True`) в случае, если файл с указанным именем не существует

Таблица 8.1 *(окончание)*

Свойство	Назначение
`CheckPathExists: Boolean`	Получает или устанавливает значение, указывающее, нужно ли выдавать предупреждающее сообщение пользователю (`True`) в случае, если он указал несуществующий путь
`DefaulExt: string`	Получает или устанавливает расширение имени файла, используемое по умолчанию
`Filter: string`	Получает или устанавливает текущую строку фильтра (маски) расширения имени файла, который определяет возможности выбора в поле **Тип файла** (Save as file type) или **Тип файлов** (Files of type)
`FilterIndex: Integer`	Получает или устанавливает номер фильтра, который используется при отображении диалога
`InitialDirectory: string`	Получает или устанавливает начальный каталог, который отображается диалогом при появлении на экране
`RestoreDirectory: string`	Получает или устанавливает значение, указывающее, восстановит ли диалог текущий каталог перед своим закрытием

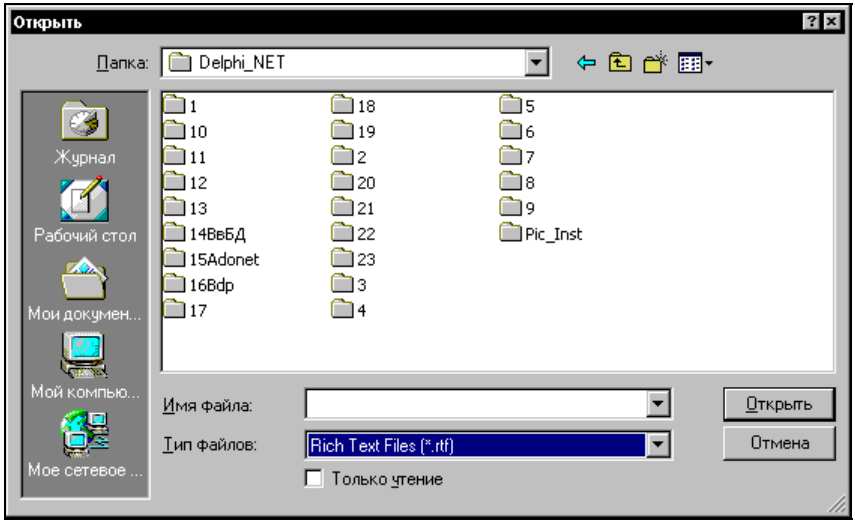


Рис. 8.2. Вид стандартного диалога открытия файла документа

Заметим, что свойство `CheckFileExists` типа `Boolean` по умолчанию имеет значение `True` для компонента `OpenFileDialog` и значение `False` — для компонента `SaveFileDialog`.

Для задания значения свойства `Filter` типа `string` в процедуре инициализации компонентов формы следует разместить, например, код вида:

```
Self.OpenDialog.Filter := 'Rich Text Files (*.rtf)|*.rtf|Text Files(*.txt' + ')
                                                                    |*.txt';
```

Заданное здесь значение фильтра обеспечивает отображение только файлов формата RTF и текстовых файлов.

Диалоги печати файлов документов

В приложениях Windows Forms при печати файлов документов используются следующие компоненты:

- `PrintPreviewDialog` (просмотр перед печатью);
- `PrintDialog` (настройка печати);
- `PageSetupDialog` (настройка параметров страницы).

Кроме того, при печати файлов документов используется компонент `PrintDocument`, который расположен на странице **Components** Палитры инструментов и принадлежит пространству имен `System.Drawing`. Данный компонент задает повторно используемый объект, который посылает выходной поток на принтер. Применение этого объекта необходимо при использовании диалогов `PrintPreviewDialog`, `PrintDialog` и `PageSetupDialog`.

Компонент `PrintPreviewDialog` задает диалоговое окно просмотра файла документа перед печатью, которое вызывается с помощью метода `ShowDialog`, вид окна приведен на рис. 8.3.

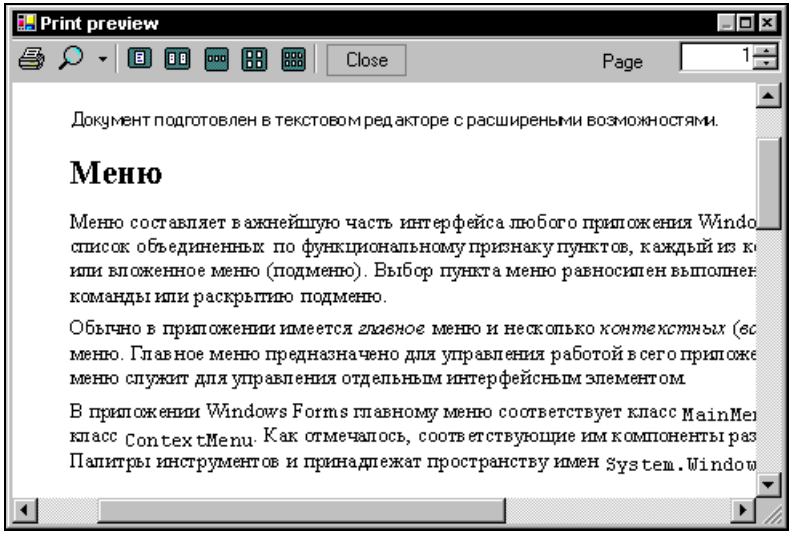


Рис. 8.3. Диалог просмотра документа перед печатью

Диалоговое окно просмотра документа перед печатью (рис. 8.3) содержит панель инструментов, с помощью кнопок которой можно выполнить печать документа, выбрать масштаб просмотра, выбрать вариант отображения нескольких страниц документа и закрыть окно.

Для компонента `PrintPreviewDialog` должно быть установлено его свойство `Document` типа `PrintDocument`. В качестве значения этого свойства указывается объект типа `PrintDocument`, который задает документ для предварительного просмотра. Связь с документом выполняется с помощью обработчика события `PrintPage` названного объекта, например, так:

```
procedure TMainForm.PrintDocument_PrintPage(sender: System.Object;
e: System.Drawing.Printing.PrintPageEventArgs);
begin
    FFirstCharOnPage := Editor.FormatRange(False, e, FFirstCharOnPage,
    Editor.TextLength);
    // проверка, есть ли еще страницы для печати
    e.HasMorePages := FFirstCharOnPage < Editor.TextLength;
end;
```

Компонент `PrintDialog` задает диалоговое окно настройки параметров печати, которое вызывается с помощью метода `ShowDialog`, вид окна приведен на рис. 8.4.

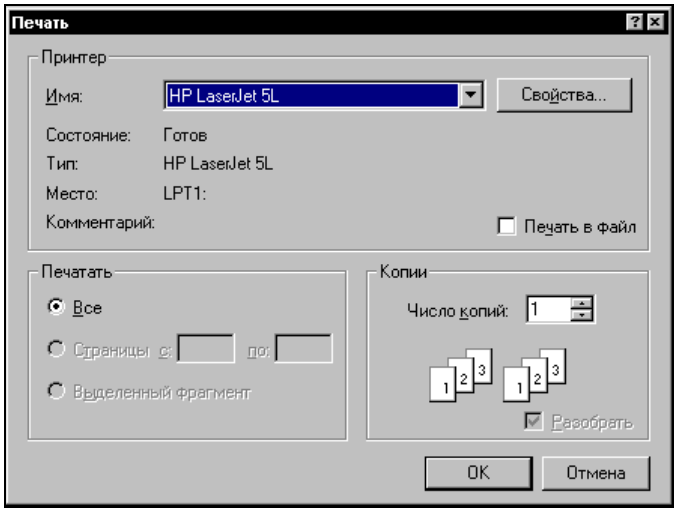


Рис. 8.4. Вид диалога настройки параметров печати

Для рассматриваемого компонента, как и для компонента `PrintPreviewDialog`, должно быть установлено его свойство `Document` типа `PrintDocument`. В качестве значения этого свойства указывается объект типа `PrintDocument`, который задает документ для печати.

Задаваемый с помощью компонента `PageSetupDialog` диалог настройки параметров страницы показан на рис. 8.5.

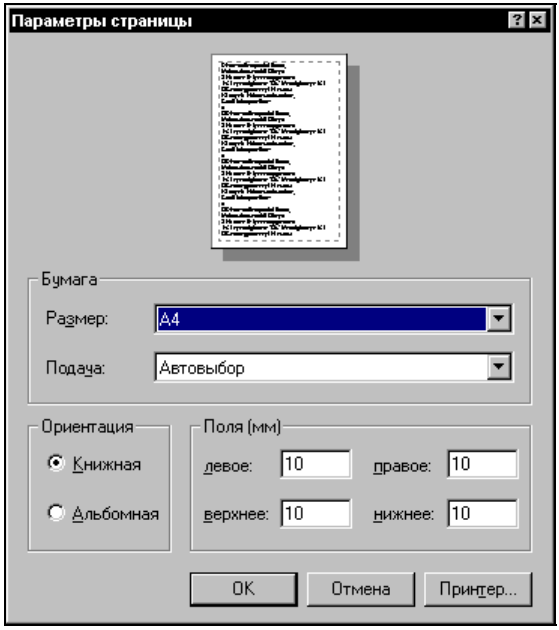


Рис. 8.5. Диалог настройки параметров страницы

Для рассматриваемого компонента, как и для компонентов `PrintPreviewDialog` и `PrintDialog`, должно быть установлено его свойство `Document` типа `PrintDocument`. В качестве значения этого свойства также указывается объект типа `PrintDocument`, который задает документ для печати.

С помощью диалога `PageSetupDialog` устанавливаются такие настройки параметров страницы документа для печати, как ориентация страницы, размеры полей, размер страницы и способ подачи бумаги.

Диалог настройки шрифта

В приложениях Windows Forms для настройки параметров шрифта документов используется компонент-диалог `FontDialog`. Вид диалога показан на рис. 8.6.

С помощью рассматриваемого диалога можно выбрать гарнитуру (поле **Шрифт**), начертание и размер шрифта. Можно задать цвет шрифта, установить зачеркнутый или подчеркнутый шрифт, а также выбрать используемый набор символов, например **Кириллица**.

Напомним, что результатом вызова рассматриваемого диалога могут быть значения **Cancel** или **OK** в зависимости от того, какая кнопка была нажата пользовате-

лем **Cancel** (Отмена) или **OK**. При этом имеется возможность принять или отклонить сделанные изменения в параметрах шрифта. Например, так, как это сделано в следующей процедуре:

```
procedure TMainForm.FormatFont;
begin
  if FontDialog.ShowDialog = System.Windows.Forms.DialogResult.OK then
  begin
    Editor.SelectionFont := FontDialog.Font;
    Editor.SelectionColor := FontDialog.Color;
  end;
end;
```

Здесь изменения в параметрах шрифта касаются выделенного фрагмента в компоненте `Editor`.

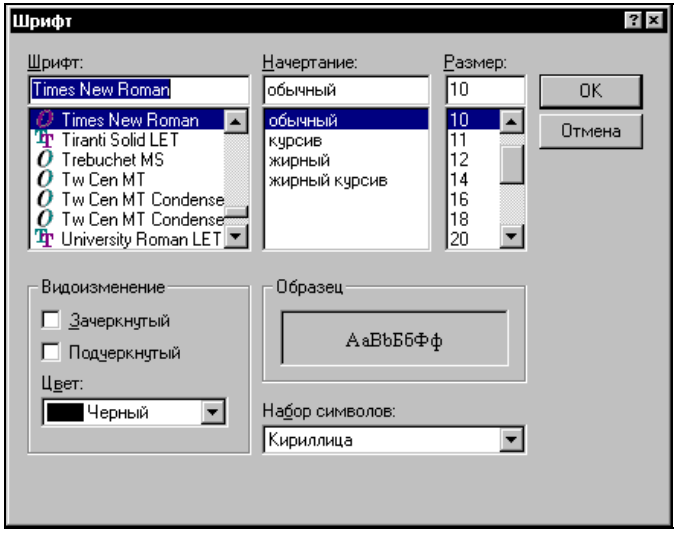


Рис. 8.6. Диалог настройки параметров шрифта

Диалог выбора цвета

В приложениях Windows Forms для выбора и настройки цвета используется компонент `ColorDialog`. Вид диалога показан на рис. 8.7.

С помощью названного компонента пользователь может выбрать нужный цвет из палитры основных цветов. Кроме того, можно выполнить визуальную настройку пользовательского цвета и добавить его в качестве одного или нескольких дополнительных цветов.

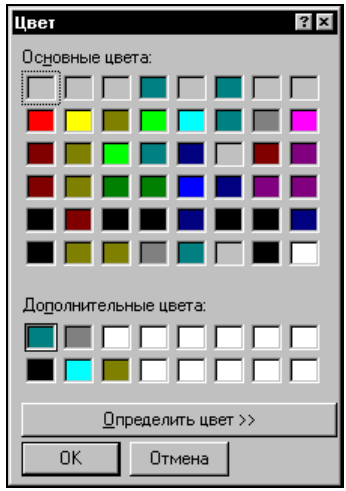


Рис. 8.7. Диалог выбора цвета

Укажем некоторые важные свойства компонента `ColorDialog`. Свойство `AnyColor` типа `Boolean` получает или устанавливает значение, указывающее, отражает ли диалог все доступные цвета базового множества цветов.

Свойство `Color` типа `Color` получает или устанавливает цвет, выбранный пользователем.

Свойство `CustomColors` типа `array of Integer` получает или устанавливает множество пользовательских цветов, показанных в диалоговом окне. Каждый элемент массива определяет пользовательский цвет как комбинацию составляющих RGB.

Панель инструментов

Панель инструментов представляет собой перемещаемое окно с "подложкой", на которой имеется набор кнопок. В приложении Windows Forms Панель инструментов представляется с помощью компонента `ToolBar`, размещенного на странице **Windows Forms** Палитры компонентов и принадлежащего пространству имен `System.Windows.Forms`.

Обычно панели инструментов размещают в верхней части главного окна приложения, в приложении может быть несколько панелей инструментов. На каждой из кнопок панели инструментов устанавливается своя картинка. Для этих целей используется компонент `ImageList` и свойство `ImageIndex` каждой кнопки. Вид панели инструментов при конструировании приложения показан на рис. 8.8.

Создание панели инструментов удобно начинать путем формирования набора изображений для кнопок с помощью компонента `ImageList`.

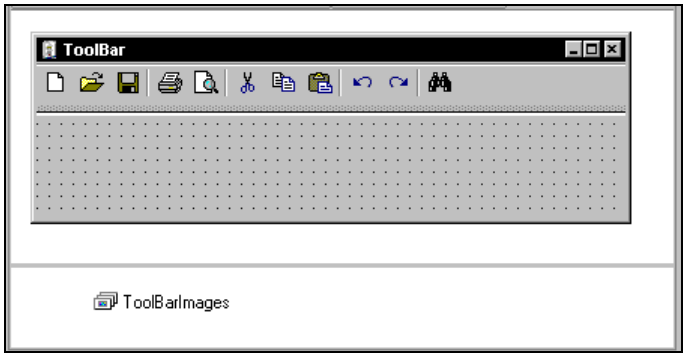


Рис. 8.8. Вид панели инструментов при конструировании приложения

Кнопки панели инструментов могут иметь три различных стиля: обычные (`PushButton`), переключатели (`ToggleButton`) и выпадающие (`DropDownButton`). Стиль кнопки панели инструментов определяется через ее свойство `Style` в составе коллекции кнопок (значения `Collection` свойства `Buttons`). Кроме того, на панели инструментов могут устанавливаться разделители (см. рис. 8.8), которые создаются как кнопки со стилем разделителя (`Separator`).

Создание и настройка свойств кнопок панели инструментов выполняется с помощью редактора ToolBarButton Collection Editor, вид которого показан на рис. 8.9. Вызов редактора выполняется двойным щелчком мыши при размещении указателя в поле значения `Collection` свойства `Buttons` компонента `ToolBar`.

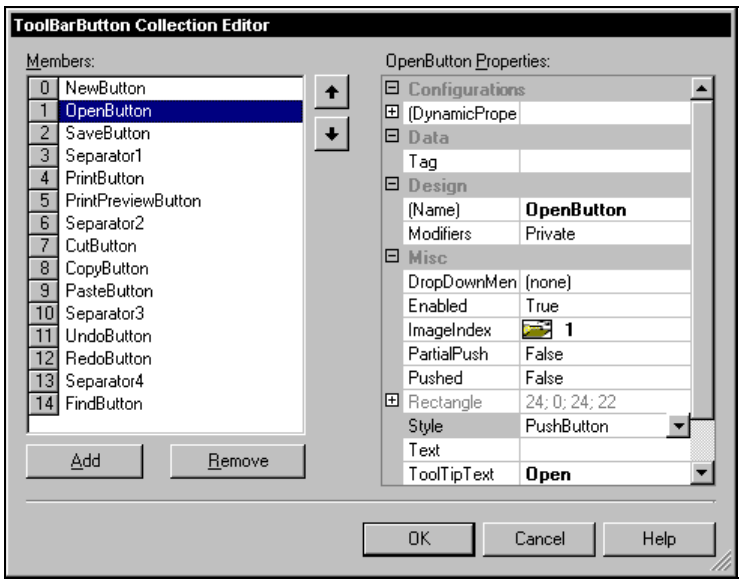


Рис. 8.9. Редактор ToolBarButton Collection Editor

Изображение на кнопке задается с помощью свойства `ImageIndex` типа `Integer`, здесь нужно указать номер картинки в составе компонента `ImageList`, нумерация которых начинается с нуля.

Свойство `Text` типа `string` позволяет задать текст на поверхности кнопки в составе панели инструментов.

С помощью свойства `ToolTipText` типа `string` для каждой кнопки можно задать текст подсказки, появляющейся при размещении указателя мыши над поверхностью этой кнопки панели инструментов.

При нажатии любой из кнопок панели инструментов возникает событие `ButtonClick`, общее для всех кнопок. Соответственно, для кнопок панели инструментов задается один общий обработчик этого события. Например, для панели инструментов, показанной на рис. 8.8, может быть задан следующий обработчик события `ButtonClick`:

```
procedure TMainForm.ToolBar_ButtonClick(sender: System.Object;
e: System.Windows.Forms.ToolBarButtonClickEventArgs);
begin
  if e.Button = NewButton then
    FileNew
  else
    if e.Button = OpenButton then
      FileOpen
    else
      if e.Button = SaveButton then
        FileSave
      else
        if e.Button = PrintButton then
          FilePrint
        else
          if e.Button = PrintPreviewButton then
            FilePrintPreview
          else
            if e.Button = CutButton then
              EditCut
            else
              if e.Button = CopyButton then
                EditCopy
              else
                if e.Button = PasteButton then
                  EditPaste
                else
                  if e.Button = UndoButton then
                    EditUndo
```

```
                    else
                      if e.Button = RedoButton then
                        EditRedo
                      else
                        if e.Button = FindButton then
                          Find;
end;
```

В приведенном обработчике производится последовательное сравнение `e.Button` с именами кнопок панели инструментов, в случае совпадения вызывается соответствующая процедура. Например, в случае нажатия кнопки с именем `FindButton` вызывается процедура `Find`.

Меню

Меню составляет важнейшую часть интерфейса любого приложения Windows. *Меню* представляет собой список объединенных по функциональному признаку пунктов, каждый из которых обозначает команду или вложенное меню (подменю). Выбор пункта меню равносилен выполнению соответствующей команды или раскрытию подменю.

Обычно в приложении имеется *главное* меню и несколько *контекстных* (*всплывающих* или *локальных*) меню. Главное меню предназначено для управления работой всего приложения, каждое из контекстных меню служит для управления отдельным интерфейсным элементом.

В приложении Windows Forms главному меню соответствует класс `MainMenu`, а контекстному меню — класс `ContextMenu`. Как отмечалось, соответствующие им компоненты размещены на странице **Dialogs** Палитры инструментов и принадлежат пространству имен `System.Windows.Forms`.

Главное меню

Для создания главного меню нужно поместить компонент типа `MainMenu` на форму и с помощью Конструктора меню задать названия пунктов и подпунктов главного меню. Затем нужно выполнить настройку свойств пунктов и подпунктов главного меню, а главное задать обработчики событий выбора пунктов и подпунктов меню. Отметим, что каждый из пунктов главного меню является объектом типа `MenuItem`, который, в свою очередь, может содержать другие объекты для реализации подменю. Вид формы и Конструктора меню в процессе создания главного меню показан на рис. 8.10. Соответствующий значок компонента `MainMenu1` отображается не на форме, а под формой.

Названия пунктов меню вводятся в верхней строке внутри прямоугольника с надписью **Type Here**. Названия подпунктов меню и вложенных подпунктов меню вводятся в прямоугольниках с такой же надписью, но расположены они в раскрывающемся списке (см. рис. 8.10).

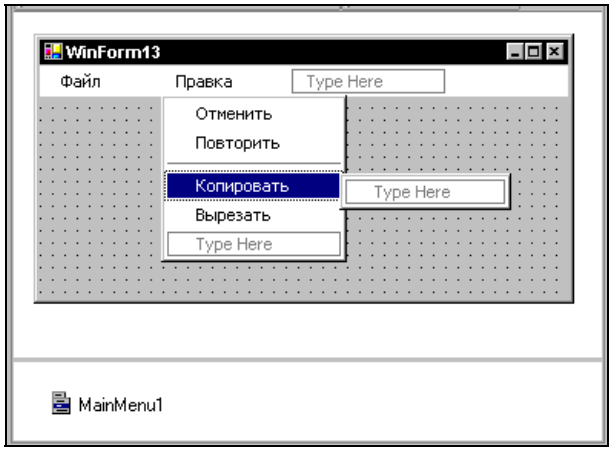


Рис. 8.10. Вид формы и Конструктора меню при создании меню

При необходимости между имеющимися пунктами меню можно вставить новый пункт меню или разделитель. К примеру, чтобы вставить разделитель над некоторым пунктом меню, достаточно выделить его щелчком мыши, нажатием правой кнопки мыши вызвать контекстное меню Конструктора меню и выбрать команду **Insert Separator** (Вставить разделитель). На рис. 8.10 разделитель вставлен между пунктами меню **Повторить** и **Копировать**. Вставка нового пункта меню выполняется аналогично с помощью контекстного меню Конструктора меню при выборе команды **Insert New** (Вставить новый). Перемещение пунктов меню выполняется с помощью команд **Cut** (Вырезать) или **Copy** (Копировать) и **Paste** (Вставить) контекстного меню Конструктора меню.

С элементами меню можно работать и программно. В частности, доступ к `MenuItem` в `MainMenu` выполняется через свойство `MenuItems`, которое получает значение типа `Menu.MenuItemCollection`, указывающее коллекцию объектов `MenuItem`, связанных с главным меню `MainMenu`. С помощью методов класса `Menu.MenuItemCollection` можно выполнять добавление и удаление объектов `MenuItem`. Например, создание объекта типа `MenuItem` с именем `MenuItemX` и добавление его с заголовком `'new'` в строку главного меню выполняется с помощью кода:

```
Self.MenuItemX:= System.Windows.Forms.MenuItem.Create;
Self.MenuItemX.Index := 0;
Self.MenuItemX.Text:='New';
MainMenu1.MenuItems.Add(MenuItemX);
```

Удаление из состава главного меню элемента с именем `MenuItemX` можно выполнить с помощью строки кода:

```
MainMenu1.MenuItems.Remove(MenuItemX);
```

После задания состава и названий всех пунктов и подпунктов главного меню нужно выполнить настройку свойств соответствующих им объектов типа `MenuIem` с помощью Инспектора объектов, а также задать обработчики событий. Укажем наиболее важные, на наш взгляд, свойства, которые требуется настраивать.

Свойство `Text` типа `string` содержит название пункта или подпункта меню, обычно оно задается при конструировании состава меню. При необходимости его можно изменить в Инспекторе объектов или программно. Последнее может потребоваться при необходимости динамической настройки меню в процессе выполнения приложения. Например:

```
MenuItem4.Text:='Название пункта';
```

Свойство `ShortCut` типа `ShortCut` получает или устанавливает значение, указывающее комбинацию клавиш, нажатие которой равносильно выбору соответствующего пункта меню. Тип `ShortCut` содержит достаточно большой набор значений, например, таких как:

- `Ins` — задает клавишу <Ins>;
- `F1` — задает клавишу <F1>;
- `Alt1` — задает комбинацию клавиш <Alt>+<1>;
- `AltF1` — задает комбинацию клавиш <Alt>+<F1>;
- `Ctrl0` — задает комбинацию клавиш <Ctrl>+<0>;
- `CtrlD` — задает комбинацию клавиш <Ctrl>+<D>.

Если в строке текста пункта или подпункта меню (свойство `Text`) поместить символ & перед некоторой буквой, то при запуске приложения эта буква в названии пункта или подпункта меню будет отображаться подчеркнутой, что сигнализирует о возможности быстрого доступа к этому пункту или подпункту меню. Для быстрого доступа к подпункту меню с помощью такой буквенной клавиши должен быть раскрыт пункт меню, который содержит этот подпункт.

Из числа событий, на которые может реагировать каждый из пунктов и подпунктов, можно выделить основное событие — `Click` (щелчок), оно возникает при выборе пункта меню с помощью мыши или посредством назначенной комбинации клавиш. Для задания обработчика этого события достаточно дважды щелкнуть мышью по пункту меню в Конструкторе меню. При этом откроется окно редактора кода, где будет показан скелет процедуры обработчика события, например, следующего вида:

```
procedure TWinForm13.MenuItem1_Click(sender: System.Object;
e: System.EventArgs);
begin
// место для кода обработки события
end;
```

Например, обработчик события выбора пункта меню, служащего для предварительного просмотра файла документа перед печатью, может выглядеть следующим образом:

```
procedure TWinForm13.FilePrintPreviewItem_Click(sender: System.Object;
e: System.EventArgs);
begin
  FilePrintPreview;
end;
```

Это означает, что в программе нами должна быть предусмотрена процедура с именем `FilePrintPreview`, содержащая код вида:

```
procedure TWinForm13.FilePrintPreview;
begin
       PrintPreviewDialog.ShowDialog;
end;
```

Здесь в процедуре выполняется вызов стандартного диалога `PrintPreviewDialog`. В принципе этот код можно разместить непосредственно в обработчике события выбора пункта меню, без создания дополнительной процедуры. Приведенный нами вариант обычно используется из тех соображений, что дополнительная процедура может содержать достаточно большой код, а вызывать ее с помощью короткого имени можно в разных местах программы. Например, вызов одной и той же процедуры в обработчике события задания пункта главного меню, пункта контекстного меню или нажатия кнопки панели инструментов.

Отметим, что хороший наглядный пример по использованию главного меню для редактора текстовых документов в формате RTF содержится в папке Program Files\Borland\Bds\4.0\Demos\Delphi.Net\WinForms\RichText. Вид формы приложения для этого примера на этапе разработки показан на рис. 8.11.

Для наглядности приведем код обработчика события выбора подпункта (команды) **Open** (Открыть) главного меню, показанного на рис. 8.11:

```
procedure TMainForm.FileOpenItem_Click(sender: System.Object;
e: System.EventArgs);
begin
  FileOpen;
end;
```

Здесь при выборе команды **Open** (Открыть) (соответствующий ей объект имеет имя `FileOpenItem`) вызывается процедура `FileOpen`. Названная процедура содержит следующий код:

```
procedure TMainForm.FileOpen;
begin
  if CanCloseCurrentFile then
  begin
    if OpenDialog.ShowDialog = System.Windows.Forms.DialogResult.OK then
```

```
    begin
      PerformFileOpen(OpenDialog.FileName);
      Editor.ReadOnly := OpenDialog.ReadOnlyChecked
        or ((System.IO.File.GetAttributes(OpenDialog.FileName) and
                          FileAttributes.ReadOnly) = FileAttributes.ReadOnly);
    end;
  end;
end;
```

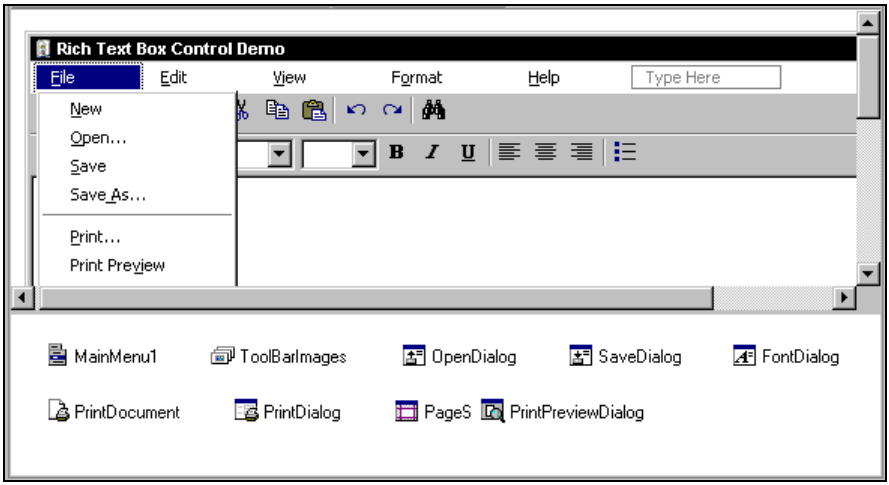


Рис. 8.11. Вид формы приложения для редактора текстовых документов

В приведенном коде требуют пояснения процедура `PerformFileOpen` и функция `CanCloseCurrentFile`. Процедура `PerformFileOpen` обеспечивает чтение содержимого файла с именем, которое нам доставляет аргумент `OpenDialog.FileName`. Эта процедура содержит следующий код:

```
procedure TMainForm.PerformFileOpen(const AFileName: string);
begin
  if (Path.GetExtension(AFileName) = '.RTF')
    or (Path.GetExtension(AFileName) = '.rtf')  then
    Editor.LoadFile(AFileName)
  else
    Editor.LoadFile(AFileName, RichTextBoxStreamType.PlainText);
  SetFileName(AFileName);
  Editor.Focus;
  Editor.Modified := False;
end;
```

С помощью функции `CanCloseCurrentFile` типа `Boolean` выполняется попытка закрыть текущий файл документа, и в случае успеха вызывается диалог открытия файла. Названная функция содержит следующий код:

```
function TMainForm.CanCloseCurrentFile: Boolean;
begin
  if Editor.Modified then
  begin
    case MessageBox.Show('Save changes to ' + FFileName + '?',
      sConfirm, MessageBoxButtons.YesNoCancel, MessageBoxIcon.Question)
    of
      System.Windows.Forms.DialogResult.Yes:
        begin
          FileSave;
          Result := True;
        end;
      System.Windows.Forms.DialogResult.No:
        Result := True;
    else
      // System.Windows.Forms.DialogResult.Cancel:
      Result := False;
    end;
  end
  else
    Result := True;
end;
```

В этой функции с помощью диалогового сообщения дается запрос пользователю на необходимость сохранения выполненных изменений в компоненте `Editor` типа `TRichTextBoxEx`, который принадлежит пространству имен: `Borland.Examples.Delphi.RichTextBoxEx`. Если пользователь выбирает вариант ответа `Cancel`, то в этом случае функция `CanCloseCurrentFile` возвращает значение `False`. При этом в процедуре `FileOpen` открытие нового файла документа не выполняется.

Отметим, что вызов процедуры `FileOpen` содержится также в процедуре-обработчике события нажатия кнопок панели инструментов:

```
procedure TMainForm.ToolBar_ButtonClick(sender: System.Object;
e: System.Windows.Forms.ToolBarButtonClickEventArgs);
begin
  if e.Button = NewButton then
    FileNew
  else
    if e.Button = OpenButton then
      FileOpen
    else
      . . .
end;
```

Здесь вызов процедуры `FileOpen` предусматривается в случае, если на панели инструментов была нажата кнопка с именем `OpenButton`.

Контекстное меню

Контекстное (всплывающее) меню появляется в случае нажатия правой кнопки мыши при условии наведения указателя мыши на форме или некотором другом элементе управления, с которым это контекстное меню связано. Из этого следует, что для каждого элемента управления можно создать свое контекстное меню, содержащее свой индивидуальный набор пунктов меню.

Подобно случаю главного меню, для создания контекстного меню нужно поместить компонент типа `ContextMenu` на форму и с помощью Конструктора меню задать названия пунктов и подпунктов. Затем нужно выполнить настройку свойств пунктов и подпунктов контекстного меню, потом задать обработчики событий выбора пунктов и подпунктов меню. Самое главное необходимо связать контекстное меню с нужным элементом управления.

Вид формы и Конструктора меню в процессе создания контекстного меню показан на рис. 8.12. Значок компонента `ContextMenu1` также отображается под формой. Процесс конструирования контекстного меню практически не отличается от конструирования главного меню.

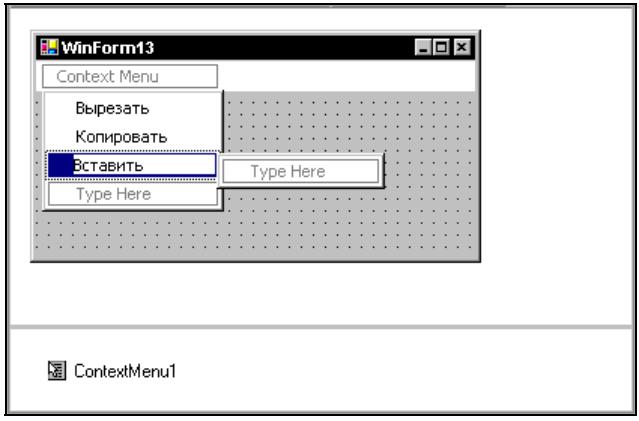


Рис. 8.12. Вид формы и Конструктора при создании контекстного меню

Для связи контекстного меню с формой или элементом управления нужно с помощью Инспектора объектов или программно их свойству `ContextMenu` типа `ContextMenu` установить в качестве значения имя нужного контекстного меню, в нашем случае — `ContextMenu1`. Отсюда следует, что одно и то же контекстное меню может быть назначено нескольким элементам управления или формам.

ЧАСТЬ III

Приложения VCL.NET

ГЛАВА 9

Визуальные компоненты

Напомним, что библиотека VCL для .NET представляет собой платформу программирования для разработки приложений в среде Delphi .NET 2006 посредством VCL. С помощью Delphi 2006 и VCL для .NET можно разрабатывать новые приложения, а также переносить имеющиеся приложения Win32 на платформу .NET, причем с сохранением большей части кода.

Компоненты библиотеки классов .NET Framework во многом повторяют основные свойства компонентов библиотеки VCL.NET. Поэтому разработка приложений VCL.NET во многом похожа на разработку приложений Windows Forms. Ввиду этого, а также в связи с ограниченным объемом книги мы рассмотрим характеристику и примеры использования компонентов библиотеки VCL.NET, которые почти не пересекаются с компонентами, рассматривавшимися нами для приложений Windows Forms.

Как отмечалось, в библиотеке VCL.NET, в отличие от .NET Framework, имена компонентов (они же имена классов) начинаются с символа `T`, в имени создаваемого объекта этот символ отсутствует. Например, компонент `TButton` (кнопка) при помещении его на форму получает имя `ButtonN`, где `N` — порядковый номер объекта, созданного с помощью этого компонента.

Страницы с визуальными компонентами

Для создания интерфейса приложений VCL.NET система Delphi .NET 2006 предлагает обширный набор визуальных компонентов, основные из которых располагаются на страницах **Standard**, **Additional** и **Win32** Палитры инструментов. Их называют *стандартными*, *дополнительными* и *32-разрядными* компонентами соответственно.

На странице **Standard** (рис. 9.1) Палитры инструментов находятся интерфейсные компоненты следующих типов, большинство из которых использовалось еще в первых версиях Windows:

- `Frames` (фреймы);
- `TMainMenu` (главное меню);
- `TPopupMenu` (всплывающее меню);
- `TLabel` (надпись);
- `TEdit` (однострочный редактор);
- `TMemo` (многострочный редактор);
- `TButton` (стандартная кнопка);
- `TCheckBox` (независимый переключатель, флажок);
- `TRadioButton` (зависимый переключатель, переключатель);
- `TListBox` (список);
- `TComboBox` (поле со списком);
- `TScrollBar` (полоса прокрутки);
- `TGroupBox` (группа);
- `TRadioGroup` (группа зависимых переключателей);
- `TPanel` (панель);
- `TActionList` (список действий).

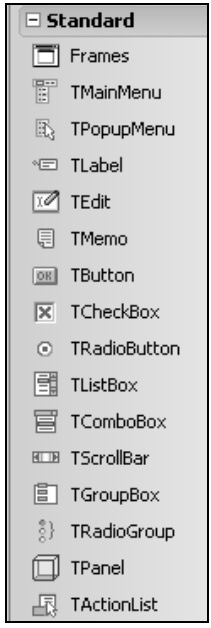


Рис. 9.1. Страница **Standard** Палитры инструментов

Компоненты перечислены последовательно в порядке расположения их значков на странице.

На странице **Additional** (рис. 9.2) Палитры инструментов находятся компоненты следующих типов:

- `TBitBtn` (кнопка с рисунком);
- `TSpeedButton` (кнопка быстрого доступа);
- `TMaskEdit` (однострочный редактор с вводом по шаблону);
- `TStringGrid` (таблица строк);
- `TDrawGrid` (таблица);
- `TImage` (графическое изображение);
- `TShape` (геометрическая фигура);
- `TBevel` (фаска);
- `TScrollBox` (область прокрутки);
- `TCheckListBox` (список флажков);
- `TSplitter` (разделитель);
- `TStaticText` (статический текст);
- `TControlBar` (контейнер для панели инструментов);
- `TApplicationEvents` (события приложения);
- `TValueListEditor` (редактор списка значений);
- `TLabeledEdit` (однострочный редактор с надписью);
- `TColorBox` (комбинированный список выбора цвета);
- `TChart` (диаграмма);
- `TActionManager` (менеджер действий);
- `TActionMainMenuBar` (главное меню действий);
- `TActionToolBar` (панель действий);
- `TXPColorMap` (цветовая карта XP для меню и панелей инструментов);
- `TStandardColorMap` (стандартная цветовая карта для меню и панелей инструментов);
- `TTwilightColorMap` (цветовая карта Twilight для меню и панелей инструментов);
- `TCustomizeDlg` (диалог настройки).

На странице **Win32** (рис. 9.3) Палитры компонентов находятся компоненты следующих типов, относящихся к 32-разрядному интерфейсу Windows:

- `TTabControl` (вкладка);
- `TPageControl` (блокнот);

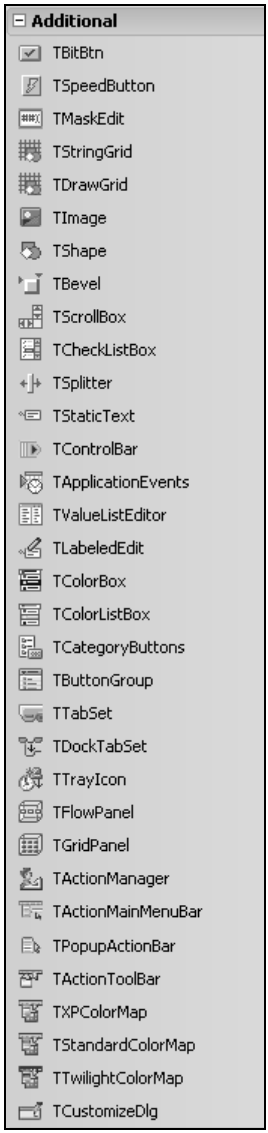


Рис. 9.2. Страница **Additional** Палитры инструментов

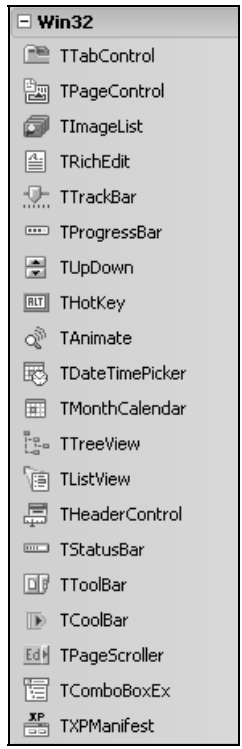


Рис. 9.3. Страница **Win32**

- `TImageList` (список графических изображений);
- `TRichEdit` (полнофункциональный текстовый редактор);
- `TTrackBar` (ползунок);
- `TProgressBar` (индикатор выполнения);

- `TUpDown` (счетчик);
- `THotKey` (редактор комбинаций горячих клавиш);
- `TAnimate` (просмотр видеоклипов);
- `TDateTimePicker` (строка ввода даты);
- `TMonthCalendar` (календарь);
- `TTreeView` (дерево объектов);
- `TListView` (список);
- `THeaderControl` (разделитель);
- `TStatusBar` (строка состояния);
- `TToolBar` (панель инструментов);
- `TCoolBar` ("крутая" панель инструментов);
- `TPageScroller` (прокрутка изображений);
- `TComboBoxEx` (расширенный комбинированный список);
- `TXPManifest` (декларация XP).

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

Базовый класс *TControl*

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

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

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

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

Такие компоненты, как редакторы типов `TEdit`, `TDBEdit`, `TMemo` или `TDBMemo`, при получении фокуса ввода отображают в своей области курсор редактирования

(текстовый курсор). По умолчанию курсор редактирования имеет вид мигающей вертикальной линии и показывает текущую позицию вставки вводимых с клавиатуры символов. Курсор редактирования перемещается с помощью клавиш управления курсором.

Компоненты, не связанные с редактированием текста, получение фокуса ввода обычно отображают с помощью пунктирного черного прямоугольника. При этом, например, для кнопки типа `Button` этот прямоугольник появляется вокруг ее заголовка, а для списков типа `ListBox` и `DBListBox` 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 элементов пользовательского интерфейса. Достоинством неоконных элементов 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.

Свойства

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

Свойство `Name` типа `TComponentName` указывает имя компонента, которое программист использует для управления компонентом во время выполнения приложения. При *динамическом* создании компонентов во время выполнения приложения они тоже автоматически получают имена по умолчанию. Разработчик может изменить имя нового компонента, программно установив нужное значение его свойства `Name`.

Приведем пример создания компонента во время выполнения программы:

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

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

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

Свойство `Align` может принимать следующие значения:

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

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

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

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

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

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

Замечание

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

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

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

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

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

Часто удобно задавать цвета с помощью констант. Отображаемый цвет зависит от параметров видеокарты и монитора, в первую очередь от установленного цветового разрешения. При использовании констант, приведенных в табл. 9.1, отображается цвет, наиболее близкий к указанному константой. Все константы, кроме `clDkGray` и `clLtGray`, можно выбирать с помощью Инспектора объектов. Дополнительно *во время выполнения* приложения можно использовать константы `clDkGray` и `clLtGray`, которые дублируют значения `clGray` и `clSilver` соответственно.

Таблица 9.1. *Константы основных цветов*

Константа	Цвет	Значение
`clAqua`	Ярко-голубой	`$FFFF00`
`clBlack`	Черный	`$000000`
`clBlue`	Синий (голубой)	`$FF0000`
`clCream`	Кремовый	`$F0FBFF`
`clFuchsia`	Пурпурный (фуксия)	`$FF00FF`
`clGray`	Серый	`$808080`
`clGreen`	Зеленый	`$008000`
`clLime`	Ярко-зеленый	`$00FF00`

Таблица 9.1 *(окончание)*

Константа	Цвет	Значение
clMaroon	Коричневый (каштановый)	$000080
clMedGray	Средне-серый	$A4A0A0
clMoneyGreen	Денежно-зеленый	$C0DCC0
clNavy	Темно-синий	$800000
clOlive	Оливковый	$008080
clPurple	Фиолетовый	$800080
clRed	Красный	$0000FF
clSilver	Серебряный	$C0C0C0
clSkyBlue	Небесно-голубой	$F0CAA6
clTeal	Сине-зеленый	$808000
clWhite	Белый	$FFFFFF
clYellow	Желтый	$00FFFF

Другой набор констант (табл. 9.2) указывает цвета в составе системной палитры Windows, установленные на вкладке **Appearance** (Оформление) диалогового окна **Display Properties** (Свойства экрана). Определяемый такой константой цвет зависит от выбранной цветовой схемы.

Таблица 9.2. *Константы системных цветов Windows*

Константа	Элемент, для которого определяется цвет
clBackground	Фон окна
clActiveCaption	Заголовок активного окна
clInactiveCaption	Заголовок неактивного окна
clMenu	Фон меню
clWindow	Фон Windows
clWindowFrame	Рамки окна
clMenuText	Текст меню
clWindowText	Текст внутри окна
clCaptionText	Текст заголовка активного окна
clInactiveCaptionText	Текст заголовка неактивного окна
clActiveBorder	Рамка активного окна
clInactiveBorder	Рамка неактивного окна

Таблица 9.2 *(окончание)*

Константа	Элемент, для которого определяется цвет
`clAppWorkSpace`	Рабочая область приложения
`clHighlight`	Фон выделенного текста
`clHightlightText`	Выделенный текст
`clBtnFace`	Кнопка
`clBtnShadow`	Тень кнопки
`clGrayText`	Неактивный интерфейсный элемент
`clBtnText`	Текст кнопки
`clBtnHighlight`	Подсвеченная кнопка
`clScrollBar`	Полоса прокрутки
`cl3DDkShadow`	Теневая сторона объемных элементов
`cl3DLight`	Яркая сторона объемных элементов
`clInfoText`	Текст информационных средств
`clInfoBk`	Фон информационных средств
`clGradientActiveCaption`	Правая сторона градиентного цвета заголовка активного окна, левую сторону цвета определяет `clActiveCaption`
`clGradientInactiveCaption`	Правая сторона градиентного цвета заголовка неактивного окна, левую сторону определяет `clInactiveCaption`
`clDefault`	Цвет по умолчанию

Свойство `Cursor` типа `TCursor` определяет *вид указателя мыши* при размещении его в области компонента. Delphi .NET 2006 предлагает более двадцати предопределенных видов указателя мыши и соответствующих им констант, основными из которых являются следующие:

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

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

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

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

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

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

Замечание

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

```
Edit1.Text := 'Иванов П.А.';
```

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

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

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

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

  Пример задания цвета компонента:

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

  Здесь для редактора `Edit1` устанавливаются зеленый цвет текста и синий цвет фона.
- Свойства `Size` и `Height` взаимозависимы, при установке значения одного из них значение второго изменяется автоматически.
- Свойства `Height` и `Width` типа `Integer` указывают соответственно вертикальный и горизонтальный *размеры* компонента в пикселах.
- Свойства `Left` и `Top` типа `Integer` определяют *координаты* левого верхнего угла компонента относительно содержащего его контейнера, например, формы или панели. Отметим, что форма также является компонентом, для нее координаты отсчитываются от левого верхнего угла экрана монитора. Свойства `Left` и `Top` совместно с `Height` и `Width` задают положение и размер компонента.
- Свойство `HelpContext` типа `THelpContext` задает *номер раздела* справочной системы. Если при выполнении программы компонент находится в фокусе ввода, то нажатие клавиши <F1> приводит к отображению на экране контекстной справки, связанной с данным компонентом.
- Свойство `Hint` типа `String` задает *текст подсказки*, появляющийся, когда курсор находится в области компонента и некоторое время неподвижен. Подсказка представляет собой поле желтого (по умолчанию) цвета с текстом, поясняющим назначение или использование компонента. Для того чтобы подсказка отображалась, следует установить значение `True` свойства `ShowHint` типа `Boolean`. По умолчанию `ShowHint` имеет значение `False`, и подсказки не отображаются.
- Свойство `PopupMenu` типа `TPopupMenu` указывает *локальное всплывающее меню*, появляющееся при размещении указателя в области компонента и одновременном нажатии правой кнопки мыши. Чтобы меню, ассоциированное с компонентом, появлялось при щелчке правой кнопкой мыши, нужно также задать значение `True` свойству `AutoPopup` типа `Boolean`. По умолчанию оно имеет значение `False`.

- Свойство `Text` типа `TCaption` содержит строку, связанную с компонентом. Значение этого свойства является содержимым компонента. Например, для компонентов `Edit` и `Memo` значение свойства `Text` отображается внутри них как редактируемые символьные данные.
- Свойство `TabOrder` типа `TTabOrder` определяет *порядок получения* компонентами контейнера фокуса при нажатии клавиши <Tab>, т. е. последовательность номеров обхода (табуляции) компонентов. По умолчанию эта последовательность определяется при конструировании формы порядком помещения компонента в контейнер: для первого компонента свойство `TabOrder` имеет значение 0, для второго — 1 и т. д. Для изменения этого порядка нужно назначить соответствующие значения свойствам `TabOrder` компонентов контейнера. Порядок табуляции компонентов в контейнере не зависит от порядка табуляции компонентов в других контейнерах.

  Два компонента не могут иметь одинаковые значения свойства `TabOrder`; система Delphi .NET 2006 следит за этим, автоматически корректируя неправильные значения. Компонент, свойство `TabOrder` которого имеет значение 0, получает управление первым.

  Свойство `TabOrder` используется совместно со свойством `TabStop` типа `Boolean`, указывающим на *возможность получения фокуса* компонентом. Если свойство `TabStop` имеет значение `True`, то элемент может получать фокус, если `False` — не может.

  Изменять порядок табуляции визуальных компонентов можно также с помощью диалогового окна **Edit Tab Order** (Изменение порядка табуляции), при этом Delphi .NET 2006 автоматически присваивает значения свойству `TabOrder` этих компонентов (рис. 9.4). Вызов диалогового окна осуществляется одноименной командой меню **Edit**.

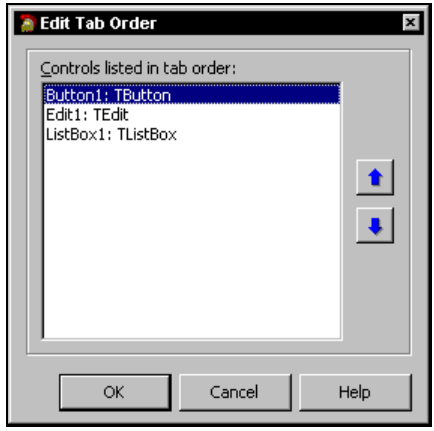


Рис. 9.4. Окно управления порядком табуляции

- Свойство `ReadOnly` типа `Boolean` определяет, разрешено ли связанному с вводом и редактированием текста элементу управления изменять находящийся в нем текст. Если свойство `ReadOnly` имеет значение `True`, то текст в элементе редактирования доступен только для чтения. Если свойство имеет значение `False` (по умолчанию), то текст можно редактировать. Запрет на редактирование относится только к пользователю, программным способом информация может быть изменена независимо от значения свойства `ReadOnly`, например, так:

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

Замечание

Даже если изменение содержимого редактора запрещено, элемент редактирования может получать фокус ввода. При получении фокуса ввода по-прежнему отображается мигающий курсор и разрешено перемещение по тексту, однако изменение содержимого редактора блокируется.

Объект поля также имеет свойство `ReadOnly`, разрешающее или запрещающее изменение его значения. Если свойству установлено значение `True`, то программисту также запрещено изменять значение поля, и при попытке это сделать генерируется исключение.

Визуальные компоненты для таких свойств, как `Color`, `Ctl3D`, `Font` и `ShowHint`, могут принимать значения соответствующих свойств родительского элемента управления, например, формы. Источники значений для указанных свойств определяются следующими свойствами-признаками типа `Boolean`:

- `ParentColor` (цвет фона);
- `ParentCtl3D` (вид компонента);
- `ParentFont` (шрифт текста);
- `ParentShowHint` (признак отображения подсказки).

Большинство этих логических свойств по умолчанию имеет значение `True`, и компонент получает значения соответствующих параметров от родителя. Подобное наследование позволяет просто и удобно изменять значения параметров для многих компонентов одновременно: для этого достаточно установить соответствующее свойство родителя в нужное значение. Например, если для формы изменить размер шрифта на значение 12, то шрифт будет изменен и для всех компонентов, расположенных 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 собственное значение, а не родительское, и при необходимости наследования программист должен снова установить значение `True` для признака наследования.

Свойство `Visible` типа `Boolean` управляет видимостью компонента. Если для него установлено значение `True`, то компонент виден пользователю, при значении `False` компонент скрыт от пользователя. Отметим, что даже если компонент не виден, им можно управлять программно. Например:

```
Edit1.Visible := True;
Edit2.Visible := False;
```

Здесь однострочный редактор типа `TEdit1` устанавливается видимым пользователю, а однострочный редактор типа `TEdit2` скрывается.

События

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

- выбор элемента управления;
- перемещение указателя мыши;
- вращение колеса мыши;
- нажатие клавиш клавиатуры;
- получение и потеря элементом управления фокуса ввода;
- перемещение объектов методом Drag and Drop (перетаскиванием).

В языке Object Pascal события также являются свойствами и принадлежат к соответствующему типу. Большинство событий носят нотификационный (уведомляющий) характер и имеют тип `TNotifyEvent`:

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

Из этого описания видно, что нотификационные события содержат только источник события, на который указывает параметр `Sender`, и больше никакой информации не несут. Существуют и более сложные события, требующие передачи дополнительных параметров, например, событие, связанное с перемещением указателя мыши, передает координаты указателя.

При выборе элемента управления возникает событие `OnClick` типа `TNotifyEvent`, которое также называют *событием нажатия*. Обычно оно возникает при щелчке мышью на компоненте. При разработке приложений событие `OnClick` является одним из наиболее часто используемых.

Приведем в качестве примера процедуру обработки события выбора элемента `Edit1`:

```
procedure TForm1.Edit1Click(Sender: TObject);
begin
Edit1.Color := Random($FFFFFF);
end;
```

Здесь при щелчке мышью в поле редактирования `Edit1` случайным образом изменяется цвет его фона.

Для некоторых компонентов событие `OnClick` может возникать и при других способах нажатия элемента управления, находящегося в фокусе ввода, например, для компонента типа `TButton` — с помощью клавиши <Space> или <Enter>, а для компонента типа `TCheckBox` — с помощью клавиши <Space>.

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

При двойном щелчке левой кнопкой мыши в области компонента, кроме того, генерируется событие `OnDblClick` типа `TNotifyEvent`. События возникают в следующем порядке: `OnMouseDown`, `OnClick`, `OnMouseUp`, `OnDblClick`, `OnMouseDown`, `OnMouseUp`.

При *перемещении указателя мыши* над визуальным компонентом непрерывно вырабатывается событие `OnMouseMove` типа `TMouseMoveEvent`. Последний описан так:

```
type TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState;
                                 X, Y: Integer) of object;
```

В обработчике события параметр `Sender` указывает, над каким элементом управления находится указатель мыши, а параметры `X` и `Y` типа `Integer` определяют координаты (позицию) указателя. Координаты задаются относительно элемента управления, определяемого параметром `Sender`. Параметр `Shift` указывает на состояние клавиш <Alt>, <Ctrl> и <Shift> клавиатуры и кнопок мыши и может принимать комбинации следующих значений:

- `ssShift` — нажата клавиша <Shift>;
- `ssAlt` — нажата клавиша <Alt>;
- `ssCtrl` — нажата клавиша <Ctrl>;
- `ssLeft` — нажата левая кнопка мыши;
- `ssMiddle` — нажата средняя кнопка мыши;
- `ssDouble` — выполнен двойной щелчок мышью.

При нажатии любой из указанных клавиш к параметру `Shift` добавляется соответствующее значение. Например, если нажата комбинация клавиш <Shift>+ +<Ctrl>, то значением параметра `Shift` является `[ssShift, ssCtrl]`. Если не нажата ни одна клавиша, то параметр `Shift` принимает пустое значение `[]`.

Рассмотрим следующий пример:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
                               X, Y: Integer);
```

```
begin
Form1.Caption := 'Координаты указателя мыши: ' + IntToStr(X) +
    ' и ' + IntToStr(Y);
end;
```

При перемещении указателя мыши в пределах формы его координаты отображаются в заголовке формы. Позиция указателя мыши отображается, если указатель находится в свободном месте формы, а не расположен над каким-либо элементом управления. Координаты `X` и `Y` отсчитываются в пикселах от левого верхнего угла формы, начиная с нуля.

При *работе с клавиатурой* генерируются события `OnKeyPress` и `OnKeyDown`, возникающие при нажатии клавиши, а также событие `OnKeyUp`, возникающее при отпускании клавиши. При нажатии клавиши возникновение событий происходит в следующем порядке: `OnKeyDown`, `OnKeyPress`, `OnKeyUp`.

При удерживании клавиши, нажатой непрерывно, генерируется событие `OnKeyDown`, событие `OnKeyUp` возникает однократно после отпускания клавиши.

Событие `OnKeyPress` типа `TKeyPressEvent` генерируется при каждом *нажатии алфавитно-цифровых* клавиш. Обычно оно обрабатывается, когда требуется реакция на нажатие одной клавиши. Тип `TKeyPressEvent` описан так:

```
type TKeyPressEvent = procedure (Sender: TObject; var Key: Char) of object;
```

Параметр `Key` содержит код ASCII нажатой клавиши, который может быть проанализирован и при необходимости изменен. Если параметру `Key` задать значение ноль (`#0`), то это соответствует отмене нажатия клавиши.

Замечание

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

В качестве примера рассмотрим обработчик события `OnKeyPress` редактора:

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

Здесь при изменении содержимого редактора `Edit1` пользователю запрещен ввод символа `!`.

Для обработки управляющих клавиш, не имеющих ASCII-кодов, можно программно использовать события `OnKeyDown` и `OnKeyUp` типа `TKeyEvent`, возникающие при нажатии любой клавиши.

Тип `TKeyEvent` описан так:

```
type TKeyEvent = procedure (Sender: TObject; var Key: Word;
Shift: TShiftState) of object;
```

Указанные события часто используются для анализа состояния управляющих клавиш <Shift>, <Ctrl>, <Alt> и др. Состояние этих клавиш и кнопок мыши указывает параметр `Shift`, который может принимать ранее рассмотренные значения. В отличие от события `OnKeyPress`, параметр `Key` имеет тип `Word`, а не `Char`, поэтому для преобразования находящегося в `Key` кода клавиши в символ можно использовать функцию `Chr()`.

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

Рассмотрим пример обработки нажатия 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 клавиши <Tab> не возникают события `OnKeyPress` и `OnKeyUp`.

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

Следующий пример демонстрирует, как производится обработка событий получения и потери фокуса элементом управления:

```
procedure TForm1.Edit1Enter(Sender: TObject);
begin
Label1.Caption := (Sender as TControl).Name + ' активен';
end;
procedure TForm1.Edit1Exit(Sender: TObject);
begin
Label1.Caption := TEdit(Sender).Name + ' не активен';
end;
```

В заголовке надписи `Label1` отображается активность (наличие или отсутствие фокуса) компонента `Edit1`. Доступ к свойству `Name` параметра `Sender` в процедурах обработки выполнен двумя способами. В первом случае параметр `Sender` с по-

мощью 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.

Методы

С визуальными компонентами, как и с другими объектами, связано большое количество методов, позволяющих создавать и удалять объекты, прорисовывать их, отображать и скрывать, а также выполнять другие операции. Рассмотрим методы, которые являются общими для всех визуальных компонентов.

Процедура SetFocus *устанавливает фокус ввода* на оконный элемент управления. Если элемент управления в данный момент времени не способен получить фокус ввода, то возникает ошибка. Поэтому при вероятном возникновении ошибки целесообразно предварительно выполнить соответствующую проверку. Проверить возможность активизации компонента позволяет функция CanFocus: Boolean, возвращающая значение True, если элемент управления может получить фокус ввода, и False — в противном случае.

Элемент управления не может получать фокус ввода, если он находится в выключенном состоянии, и его свойство Enabled имеет значение False.

Так, перед получением компонентом Edit1 фокуса ввода выполняется проверка возможности передачи ему фокуса:

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

Метод Clear служит для *очистки содержимого компонентов*, которые могут содержать текстовую информацию. Например:

```
ListBox1.Clear;
Memo1.Clear;
```

Метод Refresh используется для *обновления элемента управления*, состоящего в стирании имеющегося изображения элемента и его перерисовке. Обычно метод вызывается автоматически при необходимости перерисовки изображения. Принудительный вызов метода Refresh программным способом может понадобиться в случаях, когда программист сам управляет прорисовкой области визуального компонента, например, списка ListBox.

ГЛАВА 10

Исключения

Исключения связаны с возникновением ошибок в процессе выполнения приложений. В этой главе рассматриваются способы обработки исключений.

Виды ошибок

В процессе разработки и выполнения программы возникают ошибки:

- синтаксические;
- логические;
- динамические.

Синтаксические ошибки вызываются нарушением синтаксиса языка, они выявляются и устраняются при компиляции программы. Их обнаруживает компилятор, выдавая сообщения и указывая в тексте программы место, где возникла ошибка. Например, в условной инструкции

```
if length(Edit1.Text) = 0 then Edit1.Text = 'NoName';
```

допущена ошибка — в записи операции присваивания отсутствует знак двоеточия (:). При ее обнаружении в ходе компиляции будет выдано соответствующее сообщение (рис. 10.1).

Логические ошибки являются следствием реализации неправильного алгоритма и проявляются при выполнении программы. Их наличие обычно не приводит к выдаче пользователю каких-либо сообщений или прекращению работы всего приложения, однако программа будет работать некорректно и выдавать неправильные результаты.

Например, рассмотрим фрагмент программы, в котором вычисляется сумма 20-ти элементов массива `Matrix`:

```
sum := 1;
for n := 1 to 20 do sum := sum + Matrix[n];
```

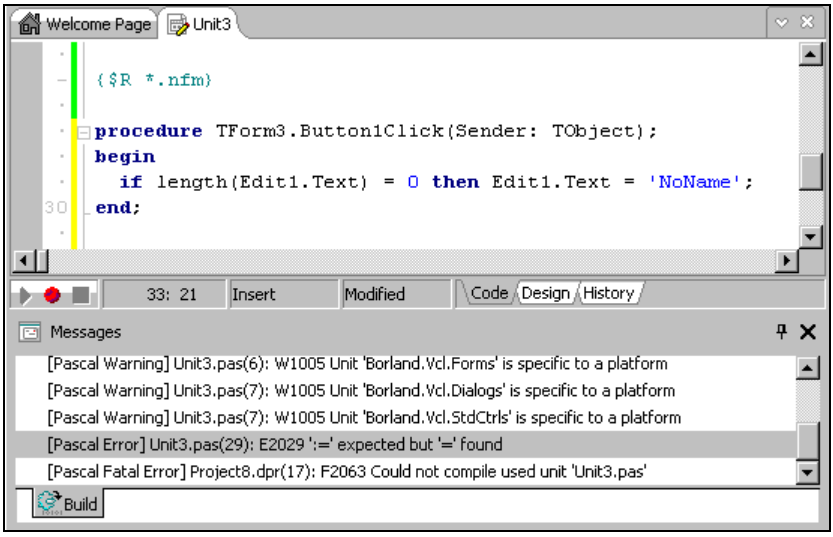


Рис. 10.1. Сообщение о синтаксической ошибке

Перед началом суммирования значение суммы `sum` должно обнуляться, однако в программе допущена ошибка: вместо нуля переменной `sum` присваивается начальное значение, равное единице. Такая ошибка не приведет к прекращению выполнения программы, однако получаемый при суммировании результат будет неверен.

Динамические ошибки возникают при выполнении программы и являются следствием неправильной работы инструкций, процедур, функций или методов программы, а также операционной системы. Динамические ошибки называют также *ошибками времени выполнения* (Runtime errors). Например, в инструкции присваивания

```
itog := count / number;
```

во время выполнения программы возможно появление ошибки, если переменная `number` будет иметь нулевое значение.

При отладке программ с целью выявления динамических ошибок удобно задавать отладочное (пошаговое или трассировочное) выполнение программы с использованием окон просмотра (**Watch List**). В окне просмотра можно указать выражения, имена переменных или свойств объекта, изменение значений которых требуется проконтролировать.

Для отладочного выполнения программы с помощью меню **Run** (Выполнение) можно использовать следующие варианты:

- команда **Step Over** (Шаг с обходом) предписывает выполнение одной строки кода программы с обходом процедур (процедура выполняется как единый модуль);

- команда **Trace Into** (Трассирование до) предписывает выполнение одной строки кода программы с заходом в процедуры и их последующим построчным выполнением;
- команда **Trace To Next Source Line** (Трассирование до следующей строки кода) предписывает выполнение программы с остановкой на следующей выполнимой строке кода программы;
- команда **Run To Cursor** (Выполнение до курсора) задает выполнение загруженной программы до места размещения курсора в Редакторе кода;
- команда **Run Until Return** (Выполнение до возврата) задает выполнение программы до момента возврата из текущей процедуры.

Для добавления очередного контролируемого выражения в окно просмотра служит команда **Run\Add Watch**. После ее выполнения открывается диалоговое окно **Watch Properties** (Свойства просмотра) для задания свойств контролируемого выражения (рис. 10.2). В поле **Expression** (Выражение) задается выражение для контроля его значения, в поле **Group name** (Имя группы) можно задать имя группы, в которую будет помещено контролируемое выражение. С помощью группы переключателей в нижней части диалогового окна выбирается формат отображения значения контролируемого выражения.

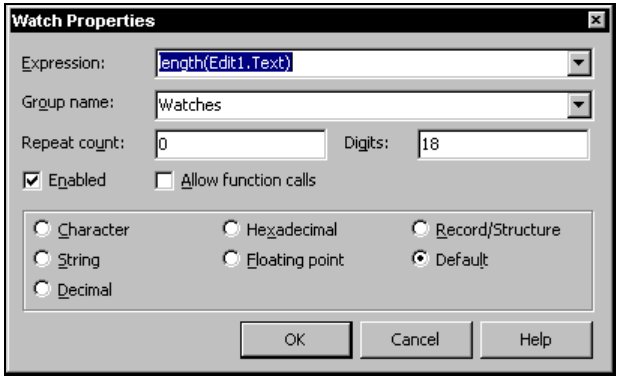


Рис. 10.2. Диалоговое окно свойств контролируемого выражения

После нажатия кнопки **OK** открывается окно **Watch List** (Список просмотра) (рис. 10.3) с добавленным к вновь созданному или существующему списку новым контролируемым выражением. Имена вкладок в окне **Watch List** соответствуют именам созданных групп просмотра. Для задания отображения окна **Watch List** (без добавления нового контролируемого выражения) служит команда **View | Debug Windows | Watches** (Вид | Окна отладки | Просмотры).

Программист должен предвидеть возможность возникновения динамических ошибок и предусматривать их обработку. Для обработки динамических ошибок введено понятие *исключения*, которое представляет собой нарушение условий

выполнения программы, вызывающее прерывание или полное прекращение ее работы. Обработка исключения состоит в нейтрализации вызвавшей его динамической ошибки.

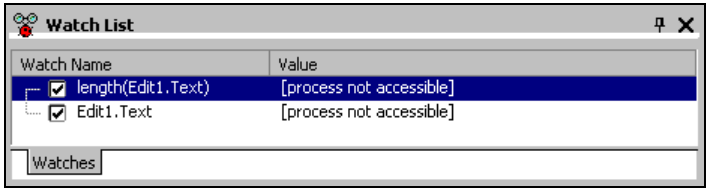


Рис. 10.3. Список контролируемых выражений в окне **Watch List**

Исключения могут возникать по различным причинам, например, в случае нехватки памяти, из-за ошибки преобразования, в результате выполнения вычислений и т. д. В любом случае, независимо от источника ошибки приложение информируется (получает сообщение) о его возникновении. Исключение остается актуальным до тех пор, пока не будет обработано глобальным обработчиком или локальными процедурами.

В Delphi .NET 2006 для обработки динамических ошибок в выполняемый файл приложения встраиваются специальные фрагменты кода, предназначенные для реагирования на исключения. Механизмы обработки ошибок инкапсулированы в класс `Exception`. Разнообразные потомки этого класса охватывают значительное число ошибок, которые могут возникнуть в среде Windows.

Возникающие при выполнении программы динамические ошибки автоматически преобразовываются средствами Delphi .NET 2006 в соответствующие объекты-исключения. Объект-исключение содержит информацию о типе ошибки и при возникновении исключения заставляет программу или ее поток (составляющую процесса) временно приостановиться. После обработки исключения объекты-исключения автоматически удаляются.

Классы исключений

Как уже сказано, исключения в Delphi .NET являются объектами. Базовым классом для всех исключений в приложениях VCL.NET служит класс `Exception`, описываемый в модуле `SysUtils`. Потомки этого класса содержат большое количество исключений, которые могут возникнуть в процессе выполнения приложения. Любые новые классы исключений должны создаваться как потомки класса `Exception`, что обеспечивает возможность их распознавания и обработки как исключений.

В отличие от других классов, имена базового класса `Exception` и его потомков, связанных с исключениями, начинаются не с буквы `T`, а с буквы `E`. Выполнять это соглашение не обязательно, но следовать ему — правило хорошего стиля.

Класс `Exception` происходит непосредственно от класса `Object` и имеет достаточно простое описание, содержащее 2 свойства и 8 методов. Так как объекты-исключения создаются при выполнении приложения, их свойства также доступны только в это время.

Свойство `Message` типа `String` содержит описание исключения. Часто при возникновении исключения этот текст появляется в диалоговом окне глобального обработчика исключений.

Свойство `HelpContext` типа `THelpContext` хранит уникальный номер (идентификатор контекста), указывающий на раздел контекстной помощи для объекта-исключения. Если этот номер отличается от нуля, то при вызове контекстной помощи отображается соответствующий раздел справки. Если же он равен нулю (по умолчанию), то отображается раздел справки родителя этого объекта.

Наиболее часто используемым методом объекта-исключения является конструктор `Create (const Msg: String)`, который создает объект-исключение. Строка `Msg` указывает текст для свойства `Message` создаваемого объекта.

Аналогичное назначение имеет конструктор `CreateFmt (const Msg: String; const Args: array of const)`, дополнительно позволяющий задать текст сообщения в форматированном виде.

Методы `CreateHelp (const Msg: String; AHelpContext: Integer)` и `CreateFmtHelp (const Msg: String; const Args: array of const; AHelpContext: Integer)` помимо создания объекта-исключения обеспечивают задание для него номера, указывающего раздел контекстной помощи (свойство `HelpContext`).

Как отмечено ранее, у класса `Exception` много потомков, каждый из которых предназначен для обработки конкретной ошибки. Некоторые классы производятся непосредственно от класса `Exception`, например, класс `EAbort`:

```
type EAbort = class(Exception);
```

Ряд классов происходит от потомков класса `Exception`. Например, класс `EMathError`, являющийся базовым классом для исключений при операциях с плавающей точкой, производится от класса `EExternal` — непосредственного потомка класса `Exception`. В свою очередь, от класса `EMathError` происходят несколько классов, обеспечивающих обработку конкретных ошибок, возникающих при выполнении операций с плавающей точкой:

```
type
EExternal = class(Exception)
  public
    ExceptionRecord: PExceptionRecord;
     end;

   EMathError  = class(EExternal);

EInvalidOp  = class(EMathError);
EZeroDivide = class(EMathError);
```

```
EOverflow   = class(EMathError);
EUnderflow  = class(EMathError);
```

Наиболее часто применяются следующие классы исключений:

- `EAbort` — "тихое" исключение, используемое для прерывания текущего блока кода без вызова глобального обработчика;
- `EOutOfMemory` — нехватка оперативной памяти для выполнения операции;
- `EInOutError` — ошибка ввода/вывода файла любого типа;
- `EIntError` — базовый класс для ошибок, связанных с целочисленными вычислениями; специализированные исключения обрабатываются потомками этого класса:
  - `EIntOverflow` — переполнение в операции с целочисленными переменными;
  - `EDivByZero` — деление целого числа на ноль;
  - `ERangeError` — присвоение целочисленной переменной значения, выходящего за пределы допустимого диапазона; может возникать при попытке обращения к элементам массива по индексу, выходящему за границы;
- `EMathError` — базовый класс для ошибок в операциях с плавающей точкой; специализированные исключения обрабатывают потомки этого класса:
  - `EInvalidOp` — ошибка в операции над числом с плавающей точкой; исключение может возникать по различным причинам, например, процессор пытается выполнить недействительную операцию или произошло переполнение стека;
  - `EZeroDivide` — деление на ноль числа с плавающей точкой;
  - `EOverFlow` — присвоение вещественной переменной значения, которое не помещается в отведенной области памяти;
  - `EUnderflow` — потеря значимости при операциях над числами с плавающей точкой, результат получает нулевое значение;
- `EInvalidPointer` — некорректная операция с указателем;
- `EInvalidCast` — неверное приведение типов с помощью инструкции `as`;
- `EConvertError` — ошибка преобразования типа, возникающая, например, при преобразовании строковых данных в числовые с помощью функции `StrToInt` или `StrToFloat`;
- `EFCreateError` — ошибка создания файла;
- `EFOpenError` — ошибка открытия файла;
- `EResNotFound` — в указанном файле отсутствует ресурс;
- `EListError`, `EStringListError` — ошибки в списках;
- `EPrinter` — ошибка печати;

- `EMenuError` — ошибка в меню приложения (часто возникает при динамической настройке меню, например, при создании нового пункта, имя которого уже существует);
- `EInvalidGraphicOperation` — неправильная операция с графическим объектом.

Потомки класса `Exception` могут добавлять к нему дополнительные свойства, конкретизирующие связанные с этими классами исключения.

Обработка исключений

Для обработки исключений в приложении есть один *глобальный* обработчик и несколько специализированных процедур-обработчиков, реагирующих на соответствующие исключения. Каждое исключение обрабатывает свой специализированный *локальный* обработчик. Исключение, не имеющее своего локального обработчика, обрабатывается глобальным обработчиком приложения.

Ниже подробно рассматривается обработка различных исключений и приводятся соответствующие примеры. В примерах других глав обработка исключений, как правило, не предусматривается с целью сокращения текста листингов.

Глобальная обработка

Механизм глобальной обработки исключений реализуется через объект `Application`, который есть в любом приложении VCL.NET. Создание и запуск объекта `Application` можно увидеть в файле проекта (dpr). Одна из функций объекта `Application` — обеспечение приложения глобальным обработчиком исключений.

При получении от операционной системы сообщения об исключении объект `Application` генерирует событие `OnException` типа `TexceptionEvent`, обработчик которого и является глобальным обработчиком исключений. По умолчанию на это событие для всех видов динамических ошибок, не имеющих своего обработчика, реагирует метод `HandleException` приложения. В теле процедуры `HandleException (Sender: TObject)` вызывается метод `ShowException` приложения, выводящий на экран диалоговое окно с описанием исключения. Такая обработка не устраняет причину исключения, но обеспечивает пользователя информацией об ошибке и облегчает ее поиск и устранение.

Программист может выполнить более полную обработку исключений, создав собственный глобальный обработчик события `OnException`. Для этого удобно использовать компонент `ApplicationEvents`.

Событие `OnException` имеет тип `TExceptionEvent`, который описан следующим образом:

```
type TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;
```

Параметр `E` представляет собой объект-исключение. С его помощью можно получить доступ к свойствам объекта-исключения, например, к свойству `Message`, содержащему описание возникшего исключения.

Рассмотрим в качестве примера процедуру-глобальный обработчик, реализующую собственную обработку исключения:

```
procedure TForm1.ApplicationEvents1Exception(Sender: TObject;
              E: Exception);
begin
MessageDlg(E.Message, mtError, [mbOK], 0);
// Здесь размещаются инструкции, выполняющие
// развитую обработку исключения
end;
```

При возникновении исключения для пользователя выводится диалоговое окно с сообщением об ошибке. Реакция на ошибку со стороны приведенной процедуры в данном случае не отличается от реакции устанавливаемого по умолчанию глобального обработчика и заключается просто в кратком информировании пользователя об ошибке. В более сложном случае глобальный обработчик может содержать код, зависящий от особенностей конкретной программы, например, освобождающий память или закрывающий рабочие файлы.

Локальная обработка

Чтобы сделать возможным использование локальных (специализированных) обработчиков исключений, в состав языка введены две конструкции: `try .. finally` и `try .. except`. Обе конструкции имеют похожий синтаксис, но разное назначение. Блоки `try` включают в себя инструкции программы (например, запись на диск или преобразование данных), при выполнении которых может возникнуть исключение.

Выбор конструкции зависит от применяемых инструкций программы и действий, выполняемых при возникновении ошибки. Конструкции `try` могут содержать одну или более инструкций, а также быть вложенными друг в друга.

Конструкция `try .. finally` состоит из двух блоков (`try` и `finally`) и имеет следующую форму:

```
try
  // Инструкции, выполнение которых может вызвать ошибку
finally
  // Инструкции, которые должны быть выполнены даже в случае ошибки
end;
```

Она применяется для выполнения всех необходимых действий перед передачей управления на следующий уровень обработки ошибки или глобальному обработчику. Такими действиями могут являться, к примеру, освобождение оперативной

памяти или закрытие файла. Эта конструкция не обрабатывает объект-исключение и не удаляет его, а выполняет действия, которые должны быть произведены даже в случае возникновения ошибки.

Конструкция `try .. finally` работает так: если в любой из инструкций блока `try` возникает исключение, то управление передается первой инструкции блока `finally`. Если же исключение не возникло, то последовательно выполняются все инструкции обоих блоков.

Рассмотрим следующий пример:

```
procedure TForm1.btnShowClick(Sender: TObject);
var MyPicture: TBitmap;
begin
try
   // Создание объекта Ris типа "растровое изображение"
   MyPicture := TBitmap.Create;
   // Загрузка изображения из указанного графического файла
   // в объект MyPicture
   MyPicture.LoadFromFile('photo.bmp');
   // Вывод изображения из объекта MyPicture на поверхность формы
   Form1.Canvas.Draw(10, 20, MyPicture);
finally
   // Даже в случае ошибки при выполнении двух предыдущих операций
   // уничтожить объект MyPicture
   MyPicture.Free;
   end;
end;
```

При нажатии кнопки `btnShow` на форме отображается изображение из файла photo.bmp. Предварительно изображение загружается из файла в промежуточный графический объект `MyPicture`. После вывода изображения на поверхность формы объект `MyPicture` уничтожается. Операция уничтожения объекта `MyPicture` и освобождения занимаемой им памяти должна быть выполнена в любом случае, поэтому инструкция `MyPicture.Free` расположена в блоке `finally`. Инструкции же, выполнение которых чревато исключением, расположены в блоке `try`. Исключение может произойти, например, при создании объекта `Ris` или при загрузке изображения из файла.

Так как конструкция `try .. finally` не ликвидирует исключительную ситуацию, в приведенной процедуре при возникновении исключения глобальный обработчик выдаст сообщение о характере ошибки.

Конструкция `try .. except` также состоит из двух блоков (`try` и `except`) и имеет следующую форму:

```
try
   {Инструкции, выполнение которых может вызвать ошибку}
```

```
except
   {Инструкции, которые должны быть выполнены в случае ошибки}
end;
```

В отличие от предыдущей, данная конструкция применяется для перехвата исключения и предоставляет возможность его обработки.

Конструкция `try .. except` работает так: если в инструкциях блока `try` возникает исключение, то управление передается первой инструкции блока `except`. Если же исключение не возникло, то инструкции блока `except` не выполняются. При появлении исключения инструкции блока `except` могут ликвидировать исключительную ситуацию и восстановить работоспособность программы. Для исключений, обрабатываемых в конструкции `try .. except`, глобальный обработчик не вызывается, а обработку ошибок должен обеспечить программист.

Рассмотрим пример локальной обработки исключения с помощью конструкции `try .. except`:

```
procedure TForm1.btnOpenClick(Sender: TObject);
begin
try
   if OpenDialog1.Execute then begin
      Table1.Active := False;
      Table1.TableName := OpenDialog1.FileName;
      Table1.Active := True;
      end;
except
   MessageDlg('Ошибка открытия таблицы ', OpenDialog1.FileName, '!',
     mtError, [mbOK], 0);
   end;
end;
```

При нажатии кнопки `btnOpen` появляется окно выбора файла таблицы базы данных для открытия. После выбора главного файла таблицы набор данных `Table1` связывается с этой таблицей, и выполняется открытие набора данных. При какой-либо ошибке, например, в случае выбора файла, не являющегося главным файлом таблицы, может возникнуть исключение. Поэтому инструкции, управляющие выбором файла и открытием набора данных, включены в блок `try`. При возникновении исключения его обрабатывают инструкции блока `except`. В данном примере обработка заключается просто в выдаче предупреждающего сообщения. Если выбор файла и открытие набора данных выполнены корректно, то инструкции блока `except` не выполняются.

Блок `except` можно разбить на несколько частей с помощью конструкций `on .. do`, позволяющих анализировать класс исключения для его более удобной и полной обработки. Конструкция `on .. do` применяется в случаях, когда действия по обработке исключения зависят от класса исключения, и имеет следующую форму:

```
on {Идентификатор: класс исключения} do
   {Инструкции обработки исключения этого класса};
else {Инструкции};
```

В инструкции `on` класс возникшего исключения сравнивается с указанным классом исключения. В случае совпадения классов выполняются инструкции после слова `do`, реализующие обработку этого исключения.

Идентификатор (произвольное имя, заданное программистом) является необязательным элементом и может отсутствовать, при этом не ставится и разделительный знак двоеточия (`:`). Идентификатор — это локальная переменная, представляющая собой экземпляр класса исключения, который можно использовать для доступа к объекту возникшего исключения. Эта переменная доступна только внутри "своей" конструкции `on .. do`.

Если класс возникшего исключения не совпадает с проверяемым классом, то выполняются инструкции после слова `else`. Блок `else` является необязательным и может отсутствовать.

Если в блоке `except` расположено несколько конструкций `on .. do`, то `else` располагается в конце блока и относится ко всей совокупности конструкций `on .. do`. Следующая после слова `else` инструкция выполняется в том случае, если обработка исключения не была осуществлена ни одной из инструкций, расположенных в любой из конструкций `do` блока. Инструкции, следующие после слов `do` и `else`, могут быть составными.

Несколько последовательных конструкций `on .. do` по структуре и логике обработки напоминают инструкцию выбора `case .. of`. Обычно в операторные скобки `on .. do` берутся исключения, появление которых наиболее вероятно.

Рассмотрим следующий пример:

```
procedure TForm1.btnAddClick(Sender: TObject);
var x, y, res :real;
begin
try
   x := StrToInt(Edit1.Text);
   y := StrToInt(Edit2.Text);
   res := x / y;
   Edit3.Text := FloatToStr(res);
except
   on EZeroDivide do begin
      MessageDlg('Попытка деления на ноль!', mtError, [mbOK], 0);
      Edit2.SetFocus;
      Edit3.Text := 'Ошибка!';
      end;
   on EO: EConvertError do begin
      MessageDlg('Ошибка преобразования!' +
                          #10#13 + EO.Message, mtError, [mbOK], 0);
```

```
      Edit1.SetFocus;
      Edit3.Text := 'Ошибка!';
      end;
   else begin
      MessageDlg('Ошибка не идентифицирована!', mtWarning, [mbOK], 0);
      Edit1.SetFocus;
      Edit3.Text := 'Ошибка!';
      end;
end;
```

В поля `Edit1` и `Edit2` вводятся два целых числа. При нажатии кнопки `btnAdd` выполняется преобразование этих чисел из текстового формата в числовой, после чего первое число делится на второе, а результат помещается в поле `Edit3`. Выполняющие эти действия инструкции могут вызвать ошибку, поэтому они помещены в блок `try`. Если возникает исключение, оно анализируется в блоке `except`. Проверяются следующие варианты: `EConvertError` (ошибка преобразования строкового типа в целочисленный) и `EZeroDivide` (деление на ноль). При этом используются две конструкции `on .. do`.

При возникновении одного из прогнозируемых исключений выдается сообщение об исключении. Любое другое исключение не будет опознано. В этом случае выполняются инструкции блока `else`. В результате выдается предупреждение о неопознанной ошибке.

В конструкции `on .. do`, применяемой для анализа исключения на ошибку преобразования, использован идентификатор `EO` (ссылка на объект-исключение), позволяющий обращаться к объекту по имени. Например, для доступа к сообщению о характере ошибки необходимо указать `EO.Message`.

Кроме выдачи сообщений, в блоке `except` выполняются такие действия, как установка фокуса ввода на определенное поле редактирования и помещение в поле результата строки `Ошибка!`.

Конструкции `try` могут быть *вложенными* и размещаться одна в другой. При этом внешняя и внутренняя конструкции могут иметь любой из двух рассмотренных видов. Обязательным условием является то, что внутренний блок должен полностью размещаться во внешнем блоке. Например:

```
try
   {Инструкции}
   try
      {Инструкции}
   finally
       {Инструкции}
   end;
except
   {Инструкции}
end;
```

или

```
try
   {Инструкции}
   try
      {Инструкции}
   except
       {Инструкции}
   end;
finally
   {Инструкции}
end;
```

Если какие-либо действия должны быть выполнены независимо от того, произошла ошибка или нет, то удобно использовать конструкцию `try .. finally`. Однако, как отмечалось, эта конструкция не обрабатывает исключение, а лишь в некоторой степени смягчает его последствия. Если же требуется произвести и локальную обработку исключения, то можно включить конструкцию `try .. finally` в конструкцию `try .. except`. При возникновении исключения это позволяет выполнить обязательные инструкции блока `finally` и обработать исключение инструкциями блока `except`. Например:

```
procedure TForm1.btnShowClick(Sender: TObject);
var MyPicture: TBitmap;
begin
try
   // Инструкции, которые могут вызвать ошибку
   try
      MyPicture := TBitmap.Create;
      MyPicture.LoadFromFile('photo.bmp');
      Form1.Canvas.Draw(10, 20, MyPicture);
   // Обязательное освобождение памяти
   finally
       MyPicture.Free;
   end;   {try .. finally}
// Анализ и обработка ошибки
except
   on EFOpenError do
      MessageDlg('Ошибка открытия файла photo.bmp!', mtError, [mbOK], 0);
   on EInOutError do
      MessageDlg('Ошибка чтения файла photo.bmp!', mtError, [mbOK], 0);
   on EInvalidGraphicOperation do
      MessageDlg('Ошибка графики!', mtError, [mbOK], 0);
   on EInvalidGraphic do
      MessageDlg('Неправильный графический формат файла!', mtError, [mbOK], 0);
   else Application.HandleException(Sender);
end;   {try .. except}
end;
```

При нажатии кнопки `btnShow` на форме отображается изображение из файла photo.bmp. В отличие от ранее рассмотренного примера, конструкция `try .. finally` вложена в конструкцию `try .. except`, в которой выполняются анализ и обработка возможного исключения. В блоке `except` проверяется, к какому классу относится возникшее исключение: ошибка открытия файла, ошибка чтения файла, ошибка выполнения графической операции или неправильный формат графического файла. Обработка возникающих исключений заключается в выдаче пользователю сообщения о характере ошибки. Если исключение не соответствует ни одному из проверяемых классов, то вызывается глобальный обработчик исключений с помощью инструкции `Application.HandleException (Sender)`, расположенной после `else`.

Аналогичным образом программируется вложение конструкции `try .. except` в конструкцию `try .. finally`.

Такие компоненты, как наборы данных `Table` и `Query`, используемые при работе с базами данных, имеют события, генерируемые при возникновении ошибок модификации данных:

- `OnEditError` (ошибка редактирования записи);
- `OnUpdateError` (ошибка обновления записи);
- `OnDeleteError` (ошибка удаления записи).

Эти события также можно использовать для локальной обработки исключений.

ГЛАВА 11

Развитые элементы интерфейса

При изучении визуальных компонентов для приложений Windows Forms нами рассмотрены наиболее простые элементы управления, предназначенные, например, для отображения текста (компонент `Label`) или однострочного редактирования данных (компонент `Edit`). В этой главе описываются более сложные элементы пользовательского интерфейса: полоса прокрутки, ползунок, счетчик, строка состояния, таблица и блокнот.

Диапазоны значений

Работа с диапазоном значений заключается в выборе и задании целочисленных значений с помощью ползунка. В приложениях VCL.NET для этого можно использовать компоненты `ScrollBar` (полоса прокрутки) и `TrackBar` (ползунок), расположенные соответственно на страницах **Standard** и **Win32** Палитры инструментов.

Оба элемента управления (рис. 11.1) представляют собой вертикально или горизонтально ориентированную полосу с ползунком. Ползунок для обоих компонентов можно передвигать с помощью мыши или клавиш управления курсором, а также клавиш <Page Up> и <Page Down>.

Обычно компонент `ScrollBar` применяется для прокрутки информации, поэтому его называют *полосой прокрутки*, или *скроллером*. Ползунок полосы прокрутки указывает относительное расстояние, на которое отображаемый фрагмент информации отстоит от ее краев. Размер ползунка может изменяться для указания видимой в настоящий момент доли информации. Управление полосой прокрутки осуществляется перетаскиванием ползунка с помощью мыши или щелчком мыши на стрелках в начале и конце полосы.

Компонент `TrackBar` называется *ползунком*, или *шкалой*, и обычно используется для изменения значений в заданном диапазоне или выбора целых чисел внутри диапазона. В Windows, например, ползунок применяется при задании размеров

Корзины, в регуляторе громкости звука, при выборе разрешения монитора и т. д. Кроме полосы, по которой он передвигается, ползунок содержит риски для отсчета значений управляемого параметра.

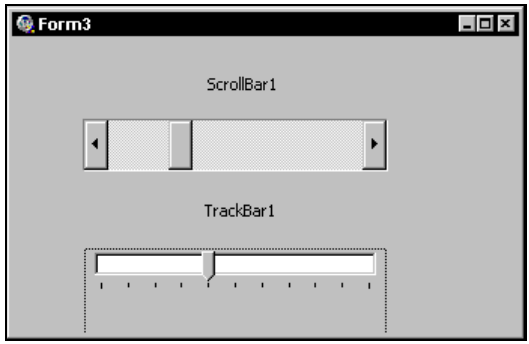


Рис. 11.1. Компоненты `ScrollBar` и `TrackBar`

Компоненты `ScrollBar` и `TrackBar` обычно используются для разных целей, но происходят от одного класса `TWinControl` и имеют схожие поведение и характеристики.

Для обоих компонентов свойства `Min` и `Max` типа `Integer` задают диапазон изменения возможных значений, а свойство `Position` типа `Integer` определяет текущую позицию ползунка в диапазоне. Пользователь изменяет значение свойства `Position`, перемещая ползунок. Допускается изменять любое из этих свойств и программно, устанавливая для них требуемые значения при разработке или при выполнении приложения. Для компонента `ScrollBar` дополнительно можно использовать метод `SetParams (APosition, AMax, AMin: Integer)`, позволяющий установить значения всех указанных свойств.

Замечание

Ни визуально, ни программно нельзя установить текущую позицию (значение свойства `Position`), выходящую за рамки допустимого диапазона (значения свойств `Min` и `Max`).

Расположение полосы компонента `ScrollBar` по горизонтали или вертикали определяет свойство `Kind` типа `TScrollBarKind`, принимающее значения:

- `sbHorizontal` (по горизонтали);
- `sbVertical` (по вертикали).

Для компонента `TrackBar` с той же целью используется свойство `Orientation` типа `TTrackBarOrientation`, принимающее значения:

- `trHorizontal` (по горизонтали);
- `trVertical` (по вертикали).

Свойства компонента `ScrollBar` `SmallChange` и `LargeChange` (типа `TScrollBarInc`) и свойства компонента `TrackBar` `LineSize` и `PageSize` (типа `Integer`) определяют *шаг перемещения* ползунка при управлении с клавиатуры. Свойства, указанные первыми, определяют шаг перемещения ползунка при использовании клавиш управления курсором, а свойства, указанные вторыми, задают шаг перемещения при использовании клавиш <Page Up> и <Page Down>. По умолчанию значения всех указанных свойств равны единице.

При визуальном или программном изменении позиции ползунка возникает событие `OnChange` типа `TNotifyEvent`. В случае визуального перемещения ползунка для компонента `ScrollBar` перед событием `OnChange` дополнительно генерируется событие `OnScroll` типа `TScrollEvent`. Тип `TScrollEvent` объявлен так:

```
type  TScrollEvent = procedure (Sender: TObject; ScrollCode: TScrollCode;
      var ScrollPos: Integer) of object;
```

Параметр `ScrollPos` определяет *позицию ползунка*, а параметр `ScrollCode` содержит *код состояния* полосы прокрутки и может принимать, к примеру, следующие значения:

- `scLineUp` — нажата клавиша <↑> или <←> либо сделан щелчок мышью на верхней (левой) стрелке полосы прокрутки;
- `scLineDown` — нажата клавиша <↓> или <→> либо сделан щелчок мышью на нижней (правой) стрелке полосы прокрутки;
- `scPageUp` — нажата клавиша <Page Up> либо сделан щелчок мышью сверху (слева) от ползунка;
- `scBottom` — ползунок перемещен мышью в крайнюю нижнюю (правую) позицию;
- `scEndScroll` — перетаскивание ползунка закончено.

События `OnChange` и `OnScroll` можно использовать для программного управления операциями, связанными с прокруткой информации. Событие `OnScroll` генерируется только при перемещении ползунка пользователем.

Ползунок `TrackBar` может содержать на шкале риски (метки), поэтому у него есть соответствующие свойства для управления их размещением. *Стиль рисок* задается свойством `TickStyle` типа `TTickStyle`, принимающим следующие значения:

- `tsAuto` (автоматическая расстановка рисок) — по умолчанию. Частота рисок задается свойством `Frequency` типа `Integer`, его значение определяет число из диапазона `Min .. Max`, которому соответствует одна метка;
- `tsManual` (риски отображаются на концах шкалы). Программно можно установить риску в любой позиции с помощью метода `SetTick (Value: Integer)`, параметр которого задает местоположение метки;
- `tsNone` (риски отсутствуют).

Риски на шкале могут находиться в различных *позициях*, определяемых свойством `TickMarks` типа `TTickMark`, которое может принимать следующие значения:

- `tmBottomRight` (риски располагаются внизу горизонтальной шкалы и справа вертикальной шкалы);
- `tmTopLeft` (риски располагаются сверху горизонтальной шкалы и слева вертикальной шкалы);
- `tmBoth` (риски располагаются по обе стороны шкалы).

Программно можно выделить внутри шкалы компонента `TrackBar` произвольный диапазон, визуально выделяемый системным цветом. Границы выделенной области определяются свойствами `SelStart` и `SelEnd` типа `Integer`, а на шкале отмечаются треугольными рисками.

Рассмотрим пример использования ползунка.

Ползунок `TrackBar1` управляет выделением текста в редакторе `Edit1` формы `Form3` (рис. 11.2). Расположенная над ползунком надпись `Label1` отражает количество символов текста, выделенных в редакторе.

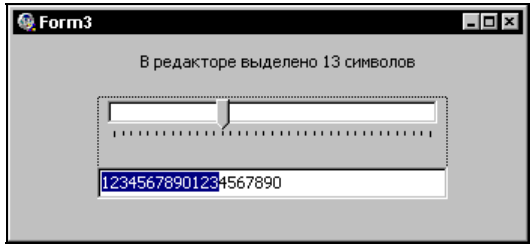


Рис. 11.2. Использование ползунка

Далее приведен текст модуля `Unit3` с описанием главной формы `Form3`.

```
unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Borland.Vcl.StdCtrls, System.ComponentModel, Borland.Vcl.ComCtrls;

type
  TForm3 = class(TForm)
    TrackBar1: TTrackBar;
    Edit1: TEdit;
    Label1: TLabel;
    procedure TrackBar1Change(Sender: TObject);
    procedure FormCreate(Sender: TObject);
```

```
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form3: TForm3;

implementation

{$R *.nfm}

procedure TForm3.FormCreate(Sender: TObject);
begin
with TrackBar1 do begin
     Min := 0;
     Max := Edit1.Width div 6;
     Position := 0;
     Frequency:=1;
     end;
 Edit1.HideSelection:=False;
end;

procedure TForm3.TrackBar1Change(Sender: TObject);
begin
Edit1.SelStart:=0;
Edit1.SelLength:=TrackBar1.Position;
Label1.Caption:='В редакторе выделено '+TrackBar1.Position.ToString+'
символов';
end;

end.
```

Выделение текста в редакторе `Edit1` выполнено в обработчике события `OnChange` ползунка с помощью свойств `SelStart` и `SelLength`. В обработчике события `OnCreate` формы выполняется настройка свойств ползунка и редактора. Значение свойства `Max` ползунка установлено с учетом того, что на один символ текста в редакторе приходится немногим больше 6 пикселов по горизонтали. Свойство `Frequency` задает единичный шаг приращения ползунка между крайними положениями. Значение `False` для свойства `HideSelection` редактора обеспечивает возможность выделения текста в редакторе внутри обработчика события для ползунка.

Реверсивные счетчики

Реверсивный счет — это увеличение или уменьшение числового значения на заданную величину (приращение). Для организации такого счета в библиотеке VCL.NET предназначены компонент `UpDown` (расположен на странице **Win32** Палитры) и компоненты `SpinButton` и `SpinEdit` (расположены на странице **Samples**), которые похожи и имеют много общих характеристик. Все эти компоненты являются *счетчиками* и имеют две кнопки со стрелками, используемыми для увеличения или уменьшения значения счетчика. Например, в окне параметров текстового процессора Microsoft Word счетчик используется для управления длиной списка запоминаемых файлов.

Компонент *UpDown*

Счетчик `UpDown` существует как автономный элемент управления и не имеет поля, в котором отображается изменяемое число; обычно он связан с другим оконным элементом управления, чаще всего с однострочным редактором `Edit` (рис. 11.3) или статическим текстом `StaticText`. Использовать для этих целей надпись `Label` нельзя, т. к. она не является оконным элементом управления. Вместо редактора `Edit` можно использовать другие компоненты, например, многострочные редакторы `Memo` и `RichEdit`, что на практике применяется редко.

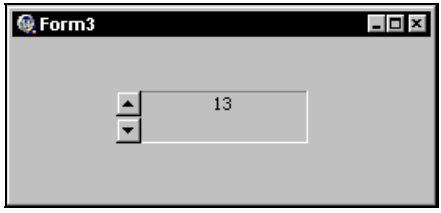


Рис. 11.3. Счетчик, ассоциированный со статическим текстом

Счетчик и ассоциированный с ним элемент взаимно дополняют друг друга и образуют парный реверсивный элемент управления: поле содержит число, а счетчик служит для уменьшения или увеличения значения числа. Счетчик `UpDown` автоматически располагается рядом со своим компонентом-"партнером". Реверсивный элемент управления не имеет заголовка, поэтому, если нужно пояснить назначение счетчика, возле него располагается надпись.

Подключение счетчика `UpDown` к ассоциированному компоненту выполняется с помощью свойства `Associate` типа `TWinControl`. В качестве парного для счетчика можно указать любой оконный элемент управления, например, `Button`, `CheckBox` или `StatusBar`. Обычно со счетчиком связывают компоненты редактирования. Сразу после установления связи между двумя компонентами счетчик выравнива-

ется относительно указанного элемента управления и размещается в его поле. Например, связать счетчик с элементом редактирования можно так:

```
UpDown1.Associate := StaticText;
```

На практике такая связь обычно устанавливается при проектировании с помощью Инспектора объектов.

Счетчик может выравниваться в поле ассоциированного компонента различными способами. *Вариант выравнивания* определяет свойство `AlignButton` типа `TUDAlignButton`, принимающее следующие значения:

- `udLeft` (по левому краю);
- `udRight` (по правому краю) — по умолчанию.

Вид счетчика определяется свойством `Orientation` типа `TUDOrientation`, принимающим следующие значения:

- `udHorizontal` (стрелки счетчика направлены вправо и влево);
- `udVertical` (стрелки счетчика направлены вверх и вниз) — по умолчанию.

Замечание

Даже в случае горизонтальной ориентации стрелок (`udHorizontal`) при управлении с клавиатуры счетчик реагирует только на нажатия клавиш <↑> и <↓>.

Числовой диапазон значения, управляемого реверсивным компонентом, задается свойствами `Min` и `Max` типа `SmallInt`. Эти свойства задают минимальное/максимальное возможные значения соответственно.

Шаг изменения значения реверсивного элемента управления при нажатии кнопки со стрелкой содержится в свойстве `Increment` типа `Integer`. По умолчанию значение этого свойства равно единице. Его можно изменять программно, устанавливая нужное целое число.

Кроме мыши, можно управлять счетчиком с помощью клавиатуры. Возможность использования *клавиатуры* зависит от свойства `ArrowKeys` типа `Boolean`. По умолчанию оно имеет значение `True`, и при нахождении реверсивного элемента в фокусе ввода значение счетчика можно изменять с помощью клавиш управления курсором (<↑> и <↓>). Если свойство `ArrowKeys` имеет значение `False`, то нажатие клавиш управления курсором не влияет на счетчик.

Текущую позицию реверсивного элемента управления определяет значение свойства `Position` типа `SmallInt`. Это значение должно находиться в диапазоне, определяемом свойствами `Min` и `Max`, независимо от того, программно или визуально оно изменяется. Когда значение свойства `Position` выходит за границу допустимого диапазона, оно фиксируется или обращается в противоположное, что зависит от свойства `Wrap` типа `Boolean`. Если это свойство установлено в `True`, то, например, при попытке пользователя увеличить текущую позицию счетчика свыше

значения Max свойство Position принимает значение Min. По умолчанию свойство Wrap имеет значение False, и значение свойства Position фиксируется при достижении границы (значения Max или Min).

Значение реверсивного счетчика содержится также в текстовом свойстве ассоциированного элемента управления, например в свойстве Text компонента Edit. Управлять текущим значением реверсивного счетчика можно с помощью обоих свойств: Position и Text, результат будет одинаков. Если значение поля участвует в *вычислениях*, то более предпочтительным является свойство Position, т. к. в этом случае не нужно преобразовывать строковое значение в числовое и обратно. В случае, когда значение счетчика используется как *строка*, удобнее применять свойство Text.

Пример использования реверсивного счетчика:

```
UpDown1.Associate := Edit1;
...
Label1.Caption := Edit1.Text;
TabControl1.TabIndex := UpDown1.Position;
```

Здесь надпись Label1 отображает значение реверсивного счетчика как строку, а элемент TabControl1 для активизации вкладки использует значение реверсивного счетчика UpDown1 как число.

В случаях, когда компонент UpDown связан с элементом редактирования, пользователь может изменять значение счетчика не только с помощью кнопок со стрелками, но и прямым редактированием с клавиатуры.

Замечание

Если ввести в поле редактора данные, которые не являются целым числом, принадлежащим заданному диапазону, то значения свойства Position счетчика и свойства Text ассоциированного с ним редактора не будут соответствовать друг другу. Свойство Text будет содержать значение, введенное пользователем, а свойство Position сохранит свое предыдущее значение. При нажатии кнопки со стрелкой счетчика информация в поле ассоциированного компонента будет приведена в соответствие со значением свойства Position.

Можно запретить непосредственное редактирование в поле парного к счетчику компонента программно, установив его свойство ReadOnly в значение True.

В текстовом представлении текущего значения счетчика (т. е. значения его свойства Position), которое отображается в ассоциированном компоненте, в качестве разделителя тысяч может использоваться запятая. Запятая появляется, если свойство Thousands типа Boolean установлено в значение True (по умолчанию), в противном случае запятая отсутствует.

При нажатии пользователем кнопок счетчика возникают события OnChanging и OnClick.

Событие OnChanging типа TUDChangingEvent генерируется при попытке *изменения значения счетчика*. Тип этого события описан следующим образом:

```
type TUDChangingEvent = procedure (Sender: TObject; var AllowChange: Boolean)
of object;
```

Параметр AllowChange позволяет принять или запретить изменение текущей позиции счетчика. Если изменение текущей позиции нежелательно, то этому параметру необходимо установить значение False.

В случаях, когда требуется проанализировать, какая из двух кнопок счетчика была нажата, можно использовать событие OnClick типа TUDClickEvent, описанного как:

```
type TUDClickEvent = procedure (Sender: TObject; Button: TUDBtnType) of object;
```

Параметр Button имеет следующие допустимые значения:

- btNext (нажата кнопка увеличения значения счетчика);
- btPrev (нажата кнопка уменьшения значения счетчика).

Компонент *SpinButton*

Счетчик SpinButton представляет собой две кнопки со стрелками. Его также можно использовать для реверсивного счета, но, по сравнению со счетчиком UpWown, он более прост и не содержит указателя текущей позиции Position и таких свойств, как Min, Max или Increment. При применении компонента SpinButton программист сам управляет размещением числа и изменением его значения. Обычно для хранения числа объявляется глобальная переменная, а изменение ее значения выполняют обработчики событий нажатия кнопок. При необходимости значение переменной можно отобразить с помощью какого-либо визуального компонента, скажем, надписи Label, а также установить значение этой переменной, например, с помощью редактора Edit.

Компонент *SpinEdit*

Счетчик SpinEdit по своему внешнему виду и функциональным возможностям соединяет в себе счетчик UpDown и ассоциированный с ним редактор Edit. Соответственно, поведение компонента SpinEdit и его характеристики являются аналогами поведения и характеристик счетчика UpDown, рассмотренного ранее. Наиболее характерными для элемента SpinEdit являются свойства Value, MinValue, MaxValue, Increment типа Integer, ReadOnly типа Boolean, событие OnChange типа TNotifyEvent.

Строка состояния

Строка состояния (статуса) представляет собой элемент управления, который отображает текущую информацию о состоянии содержимого окна и клавиатуры, контекстные подсказки по текущему пункту меню или кнопке панели инструментов и другие сведения. Строка состояния, как правило, выровнена по нижнему краю главного окна приложения. Для работы со строками состояний в VCL.NET имеется компонент `StatusBar`, находящийся на странице **Win32** Палитры инструментов. Этот компонент представляет собой строку состояния, которая может иметь одну или несколько панелей для вывода текстовой информации.

Строку состояния можно создать программно, разместив для этого в форме компонент-панель (`Panel`). Обычно эта панель не имеет заголовка и выровнена по нижнему краю формы. Рассмотрим более подробно создание строки состояния с помощью компонента `Panel`.

Внутри строки состояния размещаются несколько дополнительных панелей, служащих для отображения информации. Число таких вложенных панелей зависит от объема отображаемых сведений. Вместо панелей для вывода текущей информации можно использовать отдельный компонент, чаще всего `Label`.

Рассмотрим создание строки состояния на примере. Наша строка состояния содержит три информационных поля (рис. 11.4), отображающих текущие дату, время и координаты указателя мыши.

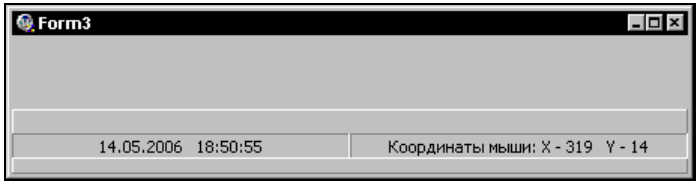


Рис. 11.4. Вид строки состояния

Далее приводится модуль `unit3` с кодом для формы `Form3`, в окне которой содержится строка состояния.

```
unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, System.ComponentModel, Borland.Vcl.StdCtrls, Borland.Vcl.ComCtrls,
Borland.Vcl.ExtCtrls;

type
  TForm3 = class(TForm)
    StatusPanel1: TPanel;
```

```
    InfPanel1: TPanel;
    InfPanel2: TPanel;
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
    procedure FormCreate(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure IdleProc(Sender :TObject; var Done :Boolean);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form3: TForm3;

implementation

{$R *.nfm}

procedure TForm3.IdleProc(Sender :TObject; var Done :Boolean);
begin
InfPanel1.Caption := DateToStr(Date)+ '   ' + TimeToStr(Time);
// Нужно для немедленного обновления  информации
Done := False;
end;

procedure TForm3.FormCreate(Sender: TObject);
begin
// Указание на обработчик свободного
// состояния приложения
Application.OnIdle := IdleProc;
// Установить значения этих свойств можно
// с помощью Инспектора объектов
StatusPanel1.Align := alBottom;
InfPanel1.BevelInner := bvNone;
StatusPanel1.BevelOuter := bvRaised;
StatusPanel1.Caption := '';
InfPanel1.Align := alNone;
InfPanel2.Align := alNone;
InfPanel1.Alignment := taCenter;
InfPanel2.Alignment := taCenter;
InfPanel1.BevelInner := bvNone;
InfPanel2.BevelInner := bvNone;
InfPanel1.BevelOuter := bvLowered;
InfPanel2.BevelOuter := bvLowered;
end;
```

```
procedure TForm3.FormMouseMove(Sender: TObject; Shift: TShiftState;
X, Y: Integer);
begin
InfPanel2.Caption := 'Координаты мыши: X - '+IntToStr(X) +
'   Y - ' + IntToStr(Y);
end;

procedure TForm3.FormResize(Sender: TObject);
var pnLength: integer;
begin
pnLength := Form3.ClientWidth div 2;
InfPanel1.Width := pnLength;
InfPanel2.Width := pnLength;
InfPanel1.Left := StatusPanel1.Left;
InfPanel2.Left := InfPanel1.Left + pnLength;
end;

end.
```

Для создания строки состояния использованы компонент-панель `StatusPanel1` и два вложенных компонента-панели `InfPanel1` и `InfoPanel2`, отображающие соответствующую информацию. Текущие координаты указателя мыши при нахождении его над формой `Form3` отслеживаются и отображаются с помощью процедуры `FormMouseMove`, вызываемой каждый раз, когда мышь перемещается.

Данные о дате и времени постоянно обновляются в процедуре `IdleProc`, которая для *приложения* является обработчиком события `OnIdle`. Для того чтобы сделанные в этой процедуре изменения отображались в строке состояния немедленно, параметру `Done` присваивается значение `False`, что предписывает системе обработать информацию, не дожидаясь других сообщений. По умолчанию параметр `Done` имеет значение `True`, и отображение времени будет обновляться только в случае прихода новых сообщений, например, при перемещении в форме указателя мыши.

Главная панель строки состояния выровнена по нижнему краю формы и при изменении размеров окна будет автоматически изменять свои размеры. Для информационных панелей пересчет их ширины и выравнивание производятся в процедуре `FormResize`.

При создании формы `Form3` в процедуре `FormCreate` определяется обработчик события `OnIdle` приложения, а также задаются основные свойства для всех панелей. Установить значения свойств панелей можно и с помощью Инспектора объектов при проектировании приложения.

Элементы с вкладками

Элемент с вкладками представляет собой элемент управления, содержащий несколько вкладок (страниц, листов), на которые пользователь может переходить, щелкая мышью на *ярлыках* этих страниц. Для работы с содержащими вкладки элементами в VCL.NET предназначены компоненты `TabControl` и `PageControl`, размещенные на странице **Win32** Палитры инструментов.

Компоненты `TabControl` и `PageControl` являются контейнерами и могут содержать другие компоненты, объединяя и группируя их. Оба компонента происходят от общего класса `TCustomTabControl` и похожи, хотя имеют и различия в поведении и использовании.

Одностраничный блокнот

Одностраничный блокнот `TabControl`, расположенный на странице **Win32** Палитры инструментов, представляет собой прямоугольную область с набором вкладок. При выборе пользователем какой-либо вкладки происходит автоматическое переключение на нее. Одностраничный блокнот обычно применяется в случаях, когда требуется набор вкладок, по функциям совпадающий с группой зависимых переключателей, например, при создании календаря или записной книжки с алфавитным указателем.

Компонент `TabControl` при отображении может иметь различные *стили*, определяемые свойством `Style` типа `TTabStyle`, принимающим следующие значения:

- `tsTabs` (стандартные вкладки объемного вида) — по умолчанию;
- `tsButtons` (вкладки в виде кнопок);
- `tsFlatButtons` (вкладки в виде плоских кнопок).

Если вкладки имеют вид кнопок, то у компонента `TabControl` отсутствует рамка, ограничивающая страницу. При необходимости программист должен сам *визуально ограничить область* указанного компонента. Для этого можно использовать такие компоненты, как `Bevel` или `Panel`, а также средства класса `TCanvas`.

Число и *названия* вкладок определяет свойство `Tabs` типа `TStrings`, представляющее собой список строк, используемый для формирования вкладок. При добавлении к списку `Tabs` новой строки автоматически создается новая вкладка с этим именем. Так как свойство `Tabs` доступно через Инспектор объектов, то при конструировании приложения вкладки создаются и настраиваются с помощью Редактора строк (String List editor). При выполнении программы работать с вкладками можно так же, как с любым объектом типа `TStrings`, настраивая свойства и вызывая соответствующие методы, например, `Add` для добавления новой вкладки, `Delete` для удаления существующей и т. д.

Так, в примере

```
TabControl1.Tabs[2] := 'Другое название';
TabControl1.Tabs.Add('Новая вкладка');
TabControl1.Tabs.Delete(7);
```

к компоненту `TabControl1` добавляется вкладка, удаляется восьмая вкладка и изменяется заголовок третьей вкладки.

Свойство `MultiLine` типа `Boolean` определяет, могут ли ярлыки вкладок отображаться в *несколько строк*. Если свойство имеет значение `False` (по умолчанию), то ярлыки выводятся одной строкой, и если они не умещаются в одной строке, то в правой части элемента управления появляются стрелки, с помощью которых можно выполнить прокрутку ярлыков.

Расположение вкладок в одностраничном блокноте определяется свойством `TabPosition` типа `TTabPosition`, которое может принимать следующие значения:

- `tpTop` (вверху) — по умолчанию;
- `tpBottom` (внизу);
- `tpLeft` (слева);
- `tpRight` (справа).

При размещении ярлыков вкладок в несколько строк, когда свойство `MultiLine` имеет значение `True`, свойство `ScrollOpposite` типа `Boolean` определяет *поведение вкладок при выборе*. При установке свойства `ScrollOpposite` в значение `True` строка с выбранной вкладкой перемещается в первый ряд. По умолчанию свойство имеет значение `False`, и перемещение вкладок не происходит. Если свойство `ScrollOpposite` устанавливается в значение `True`, то одновременно в значение `True` устанавливается и свойство `MultiLine`.

Размеры вкладок задаются свойствами `TabWidth` и `TabHeight` типа `Smallint`, определяющими их ширину и высоту в пикселах. По умолчанию оба свойства имеют значение `0`, что соответствует автоматическому изменению размеров вкладки в зависимости от заголовка.

Свойство `TabIndex` типа `Integer` *указывает выбранную вкладку* в массиве строк. Это свойство доступно для записи и может быть использовано программистом для переключения на вкладку с нужным номером. Например, для переключения на вторую вкладку можно выполнить следующую инструкцию:

```
TabControl1.TabIndex := 1;
```

Напомним, что нумерация строк в списке начинается с нуля. Если не выбрана ни одна вкладка (по умолчанию), то свойство `TabIndex` имеет значение `-1`. После динамического удаления вкладки (при выполнении программы) ни одна из вкладок не будет выбрана. Переключение вкладок при разработке приложения выполняется с помощью Инспектора объектов путем изменения значения свойства `TabIndex`.

Вкладки, кроме текста, могут отображать рисунок. Для этого нужно поместить на форму компонент `ImageList`, заполнить его рисунками, после чего указать на сформированный список рисунков значением свойства `Images` типа `TCustomImageList`. Вкладки получают рисунки из указанного списка соответственно их номерам в списке строк `Tabs`.

Все вкладки имеют одну страницу (общую область отображения), поэтому при *переключении вкладок* программист должен предусмотреть действия по обновлению информации в этой области. Для такой обработки удобно использовать событие `OnChange` типа `TNotifyEvent`, генерируемое при *активизации* вкладки, и событие `OnChanging` типа `TTabChangingEvent`, возникающее непосредственно перед активизацией.

Рассмотрим пример использования одностраничного блокнота для создания еженедельника. Для выбора дня недели в нем использован элемент `TabWeek` класса `TTabControl`. Он содержит семь ярлыков по числу дней недели, и в его области расположены две надписи: `Label1` и `Label1Date` (рис. 11.5).

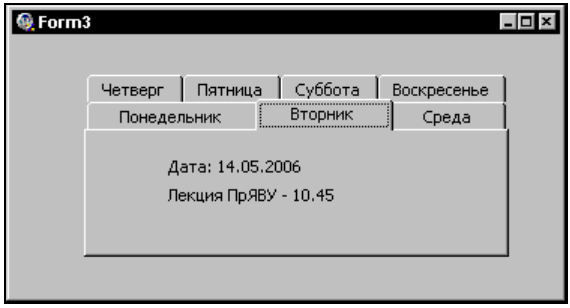


Рис. 11.5. Блокнот мероприятий недели

Далее приведен текст модуля `Unit3`, который реализует главную форму приложения.

```
unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, Borland.Vcl.StdCtrls, System.ComponentModel, Borland.Vcl.ComCtrls;

type
  TForm3 = class(TForm)
    TabWeek: TTabControl;
    Label1Date: TLabel;
    Label1: TLabel;
```

```
    procedure TabWeekChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form3: TForm3;

implementation

{$R *.nfm}

procedure TForm3.FormCreate(Sender: TObject);
begin
// Вывод текущей даты
Label1Date.Caption := 'Дата: ' + DateToStr(Date);
// Формирование вкладок и расположение ярлыков
TabWeek.Multiline := True;
TabWeek.Tabs.Clear;
TabWeek.Tabs.Add('Понедельник');
TabWeek.Tabs.Add('Вторник');
TabWeek.Tabs.Add('Среда');
TabWeek.Tabs.Add('Четверг');
TabWeek.Tabs.Add('Пятница');
TabWeek.Tabs.Add('Суббота');
TabWeek.Tabs.Add('Воскресенье');
Label1.Width:=115;
Label1.Height:=30;
Label1.WordWrap:=True;
Label1.Caption := 'Совещание 15.00';


end;

procedure TForm3.TabWeekChange(Sender: TObject);
begin
// Обновление информации в области вывода
  Label1.Caption:='';
case TabWeek.TabIndex + 1 of
  1: Label1.Caption := 'Совещание 15.00';
  2: Label1.Caption := 'Лекция ПрЯВУ - 10.45';
  3: Label1.Caption := 'Консультация - 14.40';
  4: Label1.Caption := 'Лекция КГраф - 10.45';
  5: Label1.Caption := 'Практика СУБД - 9.10';
```

```
  6: Label1.Caption := 'Бадминтон с/з - 8.30';
  7: Label1.Caption := 'Отдыхаем      ';
end;
end;

end.
```

После запуска приложения в процедуре `FormCreate` выполняется вывод текущей даты в надписи `Label1Date`, формирование вкладок и расположение ярлыков. После этого с помощью надписи `Label1` выводится информация для начальной вкладки.

При выборе другой вкладки в обработчике события `OnChange` выполняется обновление информации в области вывода компонента `TabWeek`.

Событие `OnChanging` происходит до перехода (переключения) на другую вкладку, поэтому его обработчик позволяет *запретить переключение*, если не выполнены какие-либо необходимые условия. Тип `TTabChangingEvent` этого события описан следующим образом:

```
type TTabChangingEvent = procedure (Sender: TObject; var AllowChange:
     Boolean) of object;
```

Параметр `AllowChange` определяет возможность переключения вкладки. Если `AllowChange` имеет значение `True`, то с вкладки можно перейти на другую, в противном случае — нет.

Многостраничный блокнот

Многостраничный блокнот `PageControl`, также расположенный на странице **Win32** Палитры компонентов, в отличие от компонента `TabControl`, состоит из нескольких *страниц (вкладок)*, расположенных одна под другой. Каждая страница имеет ярлык и относительно независима от других страниц. Компактное расположение страниц блокнота позволяет удобно размещать и группировать другие элементы управления. При выборе ярлыка автоматически выбирается и соответствующая страница, после чего для пользователя становятся доступными расположенные на ней элементы управления. В отличие от компонента `TabControl`, для многостраничного блокнота ярлык является частью страницы. Часто компонент `PageControl` используется для создания окон свойств, в качестве примера можно назвать окно свойств табличного процессора Microsoft Excel.

По своему внешнему виду компонент `PageControl` не отличается от компонента `TabControl` и имеет с ним некоторые общие элементы, например, свойства `Style`, `Images`, `Multiline`, `ScrollOpposite`, `HotTrack`, `TabWidth` и `TabHeight`, события `OnChange` и `OnChanging`. Но многостраничный блокнот является более сложным элементом управления, чем компонент `TabControl`. Отдельные страницы многостраничного блокнота называют также *панелями*, а сам компонент `PageControl` — *множественной панелью*.

ГЛАВА 12

Работа с графикой

Приложения Windows осуществляют вывод графической информации на экран или принтер с помощью функций GDI (Graphics Devices Interface — интерфейс графических устройств). Сама операционная система Windows является графической средой и для отображения информации также использует функции GDI.

Реализованные GDI-функции являются аппаратно-независимыми. Поэтому при выводе графической информации приложение работает не с физическим, а с логическим устройством, которое характеризуется широкой цветовой палитрой, высоким разрешением и т. п. Приложения взаимодействуют с устройствами вывода посредством драйверов, которые преобразуют аппаратно-независимые функции GDI в команды конкретного устройства.

При выполнении запроса приложения на вывод информации GDI или драйвер корректируют выводимую информацию с учетом ограниченных возможностей и особенностей физического устройства. Например, приложение может указать для цвета геометрической фигуры любой из примерно 16 миллионов цветов (2^{64}), однако далеко не любое физическое устройство обладает такими богатыми возможностями отображения цвета. Поэтому фигура будет окрашена в цвет, поддерживаемый конкретным устройством и наиболее близкий к запрошенному. Аналогичные преобразования выполняются для шрифта выводимых символов.

Такой подход позволяет приложению и операционной системе Windows функционировать относительно независимо от особенностей периферийного оборудования. Приложения Windows способны работать на компьютерах практически любой конфигурации, при этом чем лучше будут характеристики аппаратной части, тем больше качество выводимой информации будет соответствовать требованиям приложения.

Взаимодействие приложения с драйвером устройства осуществляется через специальную структуру данных, которая используется функциями GDI. Эта структура называется *контекстом отображения (дисплейным контекстом)* — DC (Display Context) и содержит основные характеристики устройства вывода, а также инструменты для рисования.

К контексту отображения относятся следующие три инструмента:

- шрифт;
- перо;
- кисть.

Программирование графики в Windows — достаточно сложный и трудоемкий процесс, VCL.NET предлагает специальные классы, существенно упрощающие использование графических средств:

- `TCanvas` (для контекста отображения);
- `TFont` (для шрифта);
- `TPen` (для пера);
- `TBrush` (для кисти).

Связанные с этими классами объекты при необходимости создаются автоматически для всех визуальных компонентов. Поэтому у них есть свойства `Canvas`, `Font`, `Pen`, `Brush`. Перечисленные свойства доступны не у всех визуальных компонентов. Например, у формы `Form` класса `TForm` доступны свойства `Canvas`, `Font`, `Brush`, у кнопки `Button` класса `TButton` — `Font`, `Brush`, у геометрической фигуры `Shape` класса `TShape` — `Font`, `Pen`, `Brush`.

Для работы с изображениями VCL.NET предлагает также классы:

- `TPicture` (контейнер для изображения);
- `TGraphic` (базовый класс для графических объектов-изображений);
- `TBitmap` (растровое изображение);
- `TIcon` (значок);
- `TMetaFile` (метафайл).

Эти классы инкапсулируются другими классами, например, `TImage`; экземпляры классов можно создавать и использовать программно.

Система Delphi .NET 2006 предоставляет возможность рисовать на поверхности компонентов в процессе выполнения приложения и создавать изображения при конструировании приложения. Есть также возможность построения диаграмм.

Возможности рисования при выполнении программы

Изображение можно строить при выполнении программы, например, рисуя на поверхности формы различными инструментами. В этом случае изображение представляет собой комбинацию графических *примитивов* (простейших фигур), таких как точка, линия, круг или прямоугольник. Наряду с графическими примитивами возможен также вывод текста.

Графические операции часто используются для программной прорисовки видимой области компонентов. Различные визуальные компоненты способны сами отображать себя и свои элементы, что чаще всего и происходит. Однако некоторые компоненты, например, списки ListBox и ComboBox, строка состояния StatusBar, таблицы DrawGrid и StringGrid, многостраничный блокнот PageControl предоставляют также возможность отображать их видимую область или определенную ее часть программно.

На *способ прорисовки* области компонента обычно указывает специальное свойство. Так, для компонента PageControl — это свойство OwnerDraw, а для компонента ListBox — свойство Style. Аналогичные свойства есть и у других элементов управления. Задав такому свойству компонента соответствующее значение, программист самостоятельно кодирует операции, связанные с выводом данных в области компонента. Обычно эти операции выполняются в процедуре — обработчике события, генерируемого при необходимости перерисовки компонента или его элемента. Например, для компонента PageControl — это событие OnDrawTab, а для компонента ListBox — событие OnDrawItem. В табл. 12.1 приведены свойства и события ряда компонентов, допускающих программную прорисовку своей области. Для свойств указаны значения, при которых выполняется обработчик события, связанного с прорисовкой области компонента.

Таблица 12.1. *Характеристики компонентов для программной прорисовки*

Компонент	Свойство	Значение	Событие
ListBox	Style	lbOwnerDrawFixed lbOwnerDrawVariable	OnDrawItem
ComboBox	Style	csOwnerDrawFixed csOwnerDrawVariable	OnDrawItem
StringGrid	DefaultDrawing	False	OnDrawCell
PageControl	OwnerDraw	True	OnDrawTab
Panels для StatusBar	Style	psOwnerDraw	OnDrawPanel

Замечание

Для большинства компонентов даже при наличии обработчика, связанного с программной прорисовкой, этот обработчик не вызывается, если соответствующее свойство компонента имеет значение, отличное от указанного в таблице.

Основной класс для связанных с рисованием графических операций — это TCanvas. С помощью его свойств и методов можно рисовать на поверхности визуальных объектов, которые включают в себя этот класс и, соответственно, имеют свойство Canvas. К ним относятся, например, такие объекты, как форма Form, надпись Label, графическое изображение Image. Наиболее часто рисование производится на поверхности формы.

Свойство `Canvas` доступно при выполнении программы, поэтому получаемые с его помощью рисунки являются *динамическими* и существуют только в процессе работы приложения. При необходимости можно сохранить рисунок в графическом файле или вывести на принтер. Создаваемые при выполнении программы рисунки могут быть неподвижными или анимационными, т. е. изменяющими свои размеры, форму и расположение.

При выполнении различных *графических операций* используются типы `TPoint` и `TRect`, описанные так:

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

Тип `TPoint` используется для задания *координат точки*, а тип `TRect` служит для определения *прямоугольной области* путем указания координат левого верхнего и правого нижнего углов.

Поверхность рисования

Поверхность рисования представляет собой объект класса `TCanvas`, иногда его называют *холстом*. У холста есть много свойств и методов, позволяющих перемещаться по поверхности рисования, отображать графические примитивы, копировать изображения и их отдельные области, а также выводить текстовую информацию.

Любая поверхность рисования включает в себя объекты "перо" `TPen`, "кисть" `TBrush` и "шрифт" `TFont`. Объекты "перо" и "кисть" используются для *прорисовки* и *заполнения* геометрических фигур, а объект "шрифт" позволяет управлять *атрибутами текста*, выводимого на поверхности.

Отметим, что компонент, имеющий свойство `Canvas`, в свою очередь сам может содержать объекты "перо", "кисть" и "шрифт" и, соответственно, иметь свойства `Pen`, `Brush` и `Font`. Таким образом, например, свойство шрифта формы `Form1.Font` не совпадает со свойством шрифта поверхности рисования формы `Form1.Canvas.Font`. Шрифт формы задает размер символов для элементов управления формы, если у них установлено значение `True` свойства `ParentFont`. Шрифт формы можно устанавливать при проектировании приложения или динамически — при его выполнении. Шрифт поверхности рисования формы определяет размер символов текста, отображаемого в форме с помощью класса `TCanvas`. Шрифт поверхности ри-

сования формы, как объект типа `TCanvas`, доступен только при выполнении программы.

При выполнении графических операций используется *текущий указатель* (указатель позиции). Он представляет собой невидимый маркер, определяющий позицию на поверхности рисования, начиная с которой выполняется следующая графическая операция. Текущая позиция определяется горизонтальной (`X`) и вертикальной (`Y`) координатами. По умолчанию начало системы координат находится в левом верхнем углу поверхности рисования, а отсчет координат осуществляется в пикселах.

Для *перемещения текущего указателя* в новую позицию можно использовать метод `MoveTo (X, Y: Integer)`. В результате выполнения этой процедуры перо устанавливается в новую позицию холста с координатами `X` и `Y`. При таком перемещении на холсте ничего не рисуется. Положение текущего указателя также изменяют методы, связанные с выводом на холст фигур и текста: при их выполнении текущий указатель остается в позиции, где завершается процесс вывода.

Для рисования геометрических фигур используются следующие методы:

- `Arc (X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)` (дуга);
- `Chord (X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)` (линия, соединяющая две точки эллипса, хорда);
- `Ellipse (X1, Y1, X2, Y2: Integer)` (эллипс с заполнением);
- `FillRect (const Rect: TRect)` (прямоугольник с заполнением);
- `FrameRect (const Rect: TRect)` (незаполненный прямоугольник, рамка);
- `LineTo (X, Y: Integer)` (линия от указателя до точки с координатами `X` и `Y`);
- `Pie (X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)` ("пирог");
- `Polygon (const Points: array of TPoint)` (многоугольник с заполнением);
- `PolyLine (const Points: array of TPoint)` (незаполненный многоугольник);
- `Rectangle (X1, Y1, X2, Y2: Integer)` (заполненный прямоугольник);
- `RoundRect (X1, Y1, X2, Y2, X3, Y3: Integer)` (заполненный прямоугольник со скругленными краями).

У методов `Arc`, `Chord` и `Ellipse` параметры `X1` и `Y1` задают координаты левого верхнего угла, а параметры `X2` и `Y2` — координаты правого нижнего угла прямоугольника, ограничивающего дугу, хорду или эллипс соответственно. Параметры `X3` и `Y3`, `X4` и `Y4` определяют координаты начальной и конечной точек дуги или хорды.

У методов `Rectangle` и `RoundRect` параметры `X1` и `Y1` задают координаты левого верхнего угла, а параметры `X2` и `Y2` — координаты правого нижнего угла прорисовываемого прямоугольника.

У методов `Polygon` и `PolyLine` параметр `Points` представляет собой массив с координатами вершин многоугольника.

У метода `Pie` параметры `X1`, `Y1`, `X2`, `Y2` задают координаты ограничивающего фигуру прямоугольника, а параметры `X3` и `Y3`, `X4` и `Y4` определяют координаты первой и второй линий радиуса соответственно.

Параметры линий фигур и их заполнение определяют текущие значения свойств пера и кисти поверхности рисования.

Рассмотрим пример рисования на поверхности формы с использованием свойства `Canvas`. Выполним вывод на поверхность формы изображения пейзажа (рис. 12.1), который создадим из эллипсов и прямоугольников. Цвет и заливка (заполнение) будут устанавливаться с помощью свойств объектов "кисть" и "перо".

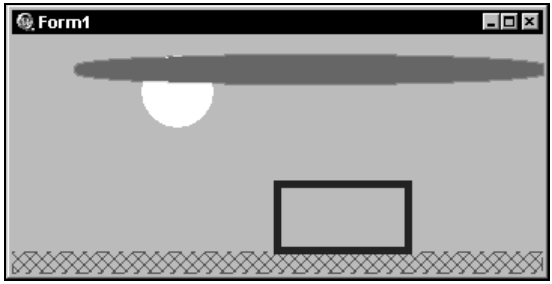


Рис. 12.1. Рисунок на поверхности формы

Исходный код модуля, реализующего вывод изображения:

```
unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure FormResize(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.nfm}

procedure TForm1.FormResize(Sender: TObject);
       var w, h, wm, hm: integer;
       begin
       Form1.Refresh;
       wm := Form1.ClientWidth;
       w := wm div 8;
       hm := Form1.ClientHeight;
       h := hm div 10;
       with Form1.Canvas do begin
            // Солнце
            Brush.Color := clYellow;
            Brush.Style := bsSolid;
            Pen.Color := clYellow;
            Ellipse(2*w, h, 3 * w, h + w);
            // Облако
            Brush.Color := clBlue;
            Brush.Style := bsSolid;
            Pen.Color := clBlue;
            Ellipse(w, h, wm, 2*h);
            // Трава
            Brush.Color := clGreen;
            Brush.Style := bsDiagCross;
            Pen.Width:=1;
            Pen.Color := clGreen;
            Pen.Style := psDashDot;
            Rectangle(0, hm - h, wm, hm);
            // Домик
            Brush.Style := bsClear;
            Pen.Width:=5;
            Pen.Color := clMaroon;
            Rectangle(4 * w, hm - 4 * h, 6 * w, hm - h);
    end;

end;

end.
```

Код, выполняющий отображение рисунка, расположен в обработчике события `OnResize` формы, поэтому при изменении ее размеров рисунок перерисовывается заново, в том числе после ее первоначального создания. Методы, выполняющие построение изображения, используют относительные (т. е. заданные относитель-

но высоты и ширины формы) координаты точек, поэтому рисунок всегда занимает всю клиентскую область формы независимо от ее размеров.

Свойство `Pen` типа `TPen` определяет *атрибуты пера*, применяемого для рисования линий и границ геометрических фигур. Управление атрибутами пера осуществляется через его свойства, основными из которых являются следующие:

- `Color` типа `TColor` (цвет пера);
- `Style` типа `TPenStyle` (стиль рисуемой линии); может принимать следующие значения:
  - `psSolid` (сплошная линия) — по умолчанию;
  - `psDash` (штриховая линия);
  - `psDot` (пунктирная линия);
  - `psDashDot` (штрихпунктирная линия);
  - `psDashDotDot` (линия вида штрих-точка-точка);
  - `psClear` (невидимая линия);
  - `psInsideFrame` (линия внутри прямоугольника поверхности рисования);
- `Width` типа `Integer` (толщина рисуемой линии в пикселах);
- `Mode` типа `TPenMode` (способ, в соответствии с которым перо при рисовании линии взаимодействует с имеющимися на холсте пикселами). Свойство `Mode` принимает следующие значения, определяющие цвет вновь рисуемой линии:
  - `pmBlack` (всегда черный);
  - `pmWhite` (всегда белый);
  - `pmNop` (не изменяется);
  - `pmNot` (инверсия цвета на поверхности рисования);
  - `pmCopy` (совпадает с цветом, указанным в свойстве `Color`) — по умолчанию;
  - `pmNotCopy` (инверсия цвета пера);
  - `pmMergePenNot` (объединение цвета пера и инверсии цвета поверхности рисования);
  - `pmMaskPenNot` (объединение общих цветов инверсии поверхности рисования и пера);
  - `pmMergeNotPen` (объединение цвета поверхности рисования и инверсии цвета пера);
  - `pmMaskNotPen` (объединение общих цветов поверхности рисования и инверсии пера);
  - `pmMerge` (объединение цвета поверхности рисования и пера);

- pmNotMerge (инверсия цвета, получаемого при значении pmMerge);
- pmMask (объединение общих цветов поверхности рисования и пера);
- pmNotMask (инверсия цвета, получаемого при значении pmMask);
- pmXor (объединение цветов, различных для поверхности рисования и пера);
- pmNotXor (инверсия цвета, получаемого при значении pmXor).

Так, в примере:

```
// Установка атрибутов пера
Form1.Canvas.Pen.Mode := pmCopy;
Form1.Canvas.Pen.Style := psDot;
Form1.Canvas.Pen.Width := 2;
Form1.Canvas.Pen.Color := clGreen;
// Рисование линии пером с установленными атрибутами
Form1.Canvas.LineTo(50, 100);
```

на поверхности формы пером рисуется зеленая пунктирная линии толщиной 2 пиксела.

Свойство Brush типа TBrush определяет *вид заполнения* геометрических фигур, например, прямоугольника или эллипса. Управление заполнением фигур осуществляется через свойства кисти, основными из которых являются следующие:

- Color типа TColor (цвет кисти);
- Style типа TBrushStyle (стиль кисти); может принимать следующие значения:
  - bsSolid (сплошная заливка);
  - bsClear (нет заливки);
  - bsHorizontal (параллельные горизонтальные линии);
  - bsVertical (параллельные вертикальные линии);
  - bsFDiagonal (параллельные диагональные линии, направленные вверх);
  - bsBDiagonal (параллельные диагональные линии, направленные вниз);
  - bsCross (прямая решетка);
  - bsDiagCross (косая решетка);
- Bitmap типа TBitmap (растровое изображение, используемое в качестве стиля кисти).

В этом примере

```
// Установка параметров кисти
Form1.Canvas.Brush.Style := bsHorizontal;
Form1.Canvas.Brush.Color := clBlue;
```

```
// Вывод эллипса, заполненного узором
Form1.Canvas.Ellipse(10, 20, 50, 100);
```

рисование осуществляется кистью: на поверхность формы выводится эллипс, заполненный параллельными горизонтальными линиями синего цвета.

Кроме стандартных узоров, задаваемых свойством `Style`, для заполнения фигур можно использовать растровое изображение из файла формата BMP. Например, в процедуре:

```
procedure TForm1.Button1Click(Sender: TObject);
var Pattern: TBitmap;
begin
// Чтение изображения из файла
Pattern := TBitmap.Create;
try
  Pattern.LoadFromFile('Pattern.bmp');
  Form1.Canvas.Brush.Bitmap := Pattern;
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Pattern.Free;
  end;
// Вывод круга, заполненного заданным изображением
Form1.Canvas.Ellipse(50, 50, 100, 100);
end;
```

в качестве заполнения фигур используется растровое изображение из файла Pattern.bmp. Изображение можно подготовить с помощью любого графического редактора.

Для определения *текущего положения пера* используется свойство `PenPos` типа `TPoint`. Например, инструкция

```
Form1.Canvas.MoveTo(Form1.Canvas.PenPos.X + 30,Form1.Canvas.PenPos.Y - 20);
```

перемещает перо на 30 пикселов вправо и на 20 пикселов вверх.

Перемещение пера можно выполнить и так:

```
Form1.Canvas.PenPos.X := Form1.Canvas.PenPos.X + 30;
Form1.Canvas.PenPos.Y := Form1.Canvas.PenPos.Y - 20;
```

Свойство `ClipRect` типа `TRect` определяет *границы области* поверхности рисования *(границы отсечения)*, в пределах которой выполняется прорисовка. Часть изображения, выходящая за пределы этого прямоугольника, отсекается и не выводится. Это свойство можно использовать для ограничения прорисовываемой области, чтобы ускорить процесс отображения. По умолчанию границы области отсечения устанавливаются по размерам поверхности компонента. Однако для компонента `Image` первоначальные размеры области отсечения равны размерам загруженного в него изображения, а для формы — размерам клиентской области.

Вывод изображения на поверхность рисования осуществляет процедура `Draw (X, Y: Integer; Graphic: TGraphic)`. В ней параметры `X` и `Y` определяют левый верхний угол начала вывода, а параметр `Graphic` задает выводимое изображение типа "растровый массив", "метафайл" или "значок". Занимаемая изображением область зависит от его размеров, и возможна ситуация, когда рисунок не помещается на холсте или закрывает другие объекты. В этом случае лучше использовать процедуру `StretchDraw (const Rect: TRect; Graphic: TGraphic)`, в которой для изображения задается не начальная точка, а прямоугольная область вывода `Rect`.

В процедуре

```
procedure TForm1.Button1Click(Sender: TObject);
        var picture: TBitmap;
        begin
        picture := TBitmap.Create;
        try
          picture.LoadFromFile('pic1.bmp');
          Form1.Canvas.Draw(10, 20, picture);
          finally
          picture.Free;
          end;
end;
```

изображение читается из файла pic1.bmp и отображается на поверхности формы.

Следующие методы и свойства предназначены для операций вывода на поверхность рисования текстовой информации.

Метод `TextOut (X, Y: Integer; const Text: String)` служит для *отображения текста* на поверхности рисования. Эта процедура выводит строку текста, заданную параметром `Text`, при этом координаты `X` и `Y` устанавливают левый верхний угол области вывода. Параметры шрифта определяются текущим значением свойства `Font` поверхности, на которой отображается текст. При вызове процедуры `TextOut` программист должен указать, что используется метод объекта `TCanvas`. В противном случае будет вызвана одноименная API-функция `TextOut`, также предназначенная для вывода текстовой информации, но отличающаяся количеством и типом параметров. Из-за этих различий при компиляции выдается сообщение об ошибке.

Пример вывода текста на поверхности формы:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
// Вывод текста на форме
Canvas.TextOut(100, 100, 'Тект');
// Другой вариант инструкции вывода текста на форме
// Form1.Canvas.TextOut(100, 100, 'Тект');
```

```
// Этот вызов приведет к ошибке компиляции:
// TextOut(100, 100, 'Text');
end;
```

Вывод текста на холст можно выполнить также с помощью метода `TextRect`, отличающегося от метода `TextOut` наличием дополнительного параметра `Rect`, который ограничивает область вывода.

В ряде случаев для *управления размещением текста* на поверхности рисования требуется определить размеры прямоугольника, занимаемого выводимой строкой. Для этого удобно использовать методы `TextHeight`, `TextWidth` и `TextExtent`.

Функции `TextHeight (const Text: String): Integer` и `TextWidth (const Text: String): Integer` возвращают в качестве результирующих значений (в пикселах) соответственно *высоту* и *ширину* прямоугольной области, занимаемой строкой `Text`.

Функция `TextExtent (const Text: String): TSize` возвращает запись, содержащую значение (в пикселах) высоты `cy` и ширины `cx` области вывода. Тип записи `TSize` описан так:

```
type  TSize = record
    cx: Longint;
    cy: Longint;
    end;
```

Размеры области вывода текста зависят от шрифта и его параметров. Например, при увеличении размера шрифта (свойство `Canvas.Font.Size`) соответственно увеличиваются высота и ширина области вывода.

Важным свойством объекта класса `TCanvas` является свойство `Pixels [X,Y: Integer]` типа `TColor`, которое определяет цвет пиксела в точке, указанной индексами `X` и `Y` двумерного массива координат `Pixels`. Это свойство доступно для чтения и для записи.

Пример использования свойства `Pixels`:

```
procedure TForm1.Button1Click(Sender: TObject);
var n :integer;
begin
    for n := 0 to Form1.ClientWidth do
    Form1.Canvas.Pixels[n,20] := clRed;
end;
```

Приведенная процедура выполняет вывод на поверхность формы красной горизонтальной линии на всю ширину формы.

Объекты класса `TCanvas` включают в себя объекты `TFont`, `TPen` и `TBrush` и, соответственно, имеют свойства `Font`, `Pen` и `Brush`. Эти объекты могут также входить в состав других визуальных компонентов, например, формы или панели.

Свойство `Font` типа `TFont` устанавливает *параметры шрифта*, применяемого для отображения текста на поверхности рисования. Управление параметрами шрифта осуществляется через его свойства, основными из которых являются следующие:

- `Name` типа `TFontName` (название шрифта, например, Arial или Times New Roman). Отметим, что свойство `Name` шрифта не связано с одноименным свойством самого компонента;
- `Size` типа `Integer` (размер шрифта в пунктах). Пункт равен 1/72 дюйма;
- `Height` типа `Integer` (размер шрифта в пикселах). Если значение этого свойства является положительным числом, то в него включен и межстрочный интервал. Если размер шрифта имеет отрицательное значение, то интервал не учитывается;
- `Style` типа `TFontStyle` (стиль шрифта), может принимать комбинации следующих значений:
  - `fsItalic` (курсив);
  - `fsBold` (полужирный);
  - `fsUnderline` (подчеркнутый);
  - `fsStrikeOut` (перечеркнутый);
- `Color` типа `TColor` (цвет выводимого текста).

Свойства `Size` и `Height` являются взаимозависимыми: при установке значения одного из них второе свойство автоматически получает соответствующее значение.

Так, в примере

```
// Установка параметров шрифта
Form1.Canvas.Font.Name := 'Courier';
Form1.Canvas.Font.Style := [fsBold] + [fsUnderline];
Form1.Canvas.Font.Size := 14;
Form1.Canvas.Font.Color := clRed;
// Вывод текста шрифтом с установленными параметрами
Form1.Canvas.TextOut(100, 100, 'Управление параметрами шрифта');
```

на поверхность формы выводится полужирный подчеркнутый текст шрифтом Courier размером 14 пунктов с красным цветом символов.

Для получения *списка доступных* (установленных в системе) *шрифтов* можно использовать свойство `Fonts` глобального объекта `Screen` (экрана).

Для выбора и установки параметров шрифта удобно использовать стандартное диалоговое окно `FontDialog`.

При *изменении содержимого поверхности* рисования генерируются события `OnChanging` и `OnChange` типа `TNotifyEvent`. Первое из них возникает *перед* измене-

нием поверхности, а второе — *после*. Отметим, что эти события генерируются только при использовании методов, связанных с рисованием и выводом текста, перемещение указателя с помощью метода `MoveTo` событиями `OnChanging` и `OnChange` не сопровождается.

Так как свойство `Canvas` на этапе проектирования недоступно, обработчики его событий программируются вручную.

Свойством `Canvas` обладают все визуальные компоненты, но не для всех визуальных компонентов это свойство программно доступно (описано со спецификатором доступа `Public`). В случае, когда требуется рисование на поверхности некоторого компонента, а его свойство `Canvas` недоступно, можно применить один из следующих приемов:

- разместить в данном компоненте другой элемент, который предоставляет свойство `Canvas`, например, `Image` или `PaintBox`, и рисовать на этом элементе;
- использовать специальную переменную типа `TCanvas`, которую следует через ее свойство `Handle` связать с контекстом отображения данного компонента, и выполнять рисование на поверхности связанного с этой переменной объекта.

При выполнении различных графических операций часто используются тип `TColor` и соответствующее ему свойство `Color`, предназначенные для *управления цветом* объектов. Для управления цветом со стороны пользователя удобно применять стандартное диалоговое окно `ColorDialog`.

Для работы со *значениями цветов* программист может использовать специальные GDI-функции (например, `ColorToRGB`, `GetRValue`, `GetGValue`, `GetBValue` и `RGB`), предназначенные для получения и преобразования цвета и его составляющих (красной, зеленой и синей).

Кроме цветовых, в распоряжении программиста имеется множество других GDI-функций, связанных с графическими операциями. В параметрах этих функций используется свойство `Handle` объекта `TCanvas`, которое представляет собой дескриптор контекста устройства, в данном случае поверхности рисования.

Графические компоненты

При конструировании формы для создания визуальных эффектов и изображений можно использовать соответствующие компоненты. Действия, связанные с построением изображения, напоминают работу в среде графического редактора.

Подобным образом можно создавать относительно простые визуальные эффекты, такие как отображение рамок, фасок или элементарных геометрических фигур. Это связано с тем, что набор компонентов, из которых конструируется изображение, невелик, а возможности этих компонентов ограниченны. Наиболее часто

используются такие графические компоненты, как геометрическая фигура (`Shape`), фаска (`Bevel`), графическое изображение (`Image`).

Графические элементы являются *неоконными визуальными компонентами* и происходят от класса `TGraphicControl`.

Компонент *Shape*

Для отображения геометрических фигур в VCL.NET служит компонент `Shape` типа `TShape`, который расположен на странице **Additional** Палитры инструментов. *Вид фигуры* (рис. 12.2), отображаемой этим компонентом, определяется одноименным свойством `Shape` типа `TShapeType`, принимающим следующие значения:

- `stCircle` (круг);
- `stEllipse` (эллипс);
- `stRectangle` (прямоугольник);
- `stRoundRect` (прямоугольник со скругленными углами);
- `stRoundSquare` (квадрат со скругленными углами);
- `stSquare` (квадрат).

Управление *цветом* и *заполнением* фигуры выполняется с помощью свойств `Pen` и `Brush`.

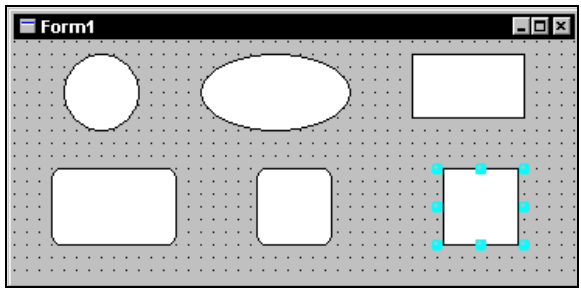


Рис. 12.2. Фигуры, отображаемые компонентом `Shape`

Компонент *Bevel*

Фаски представляют собой прямоугольные области, рамки и линии. Они имеют плоский или объемный вид и обычно используются для визуального выделения других элементов формы с целью более наглядного их восприятия. Для работы с фаской в VCL.NET служит компонент `Bevel`, который расположен на странице **Additional** Палитры инструментов.

Фигура (рис. 12.3), используемая для фаски, задается свойством `Shape` типа `TBevelShape`, принимающим следующие значения:

- `bsBottomLine` (линия снизу);
- `bsBox` (прямоугольник);
- `bsFrame` (рамка);
- `bsLeftLine` (линия слева);
- `bsRightLine` (линия справа);
- `bsSpacer` (прямоугольная область, невидимая при выполнении программы);
- `bsTopLine` (линия сверху).

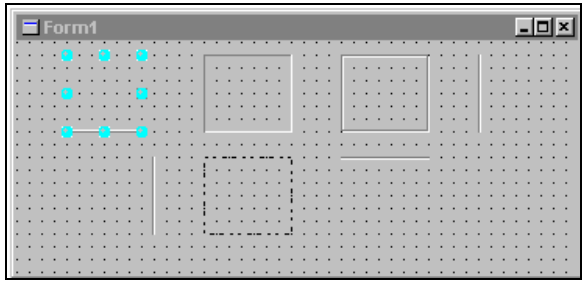


Рис. 12.3. Виды фасок, отображаемых компонентом `Bevel`

Свойство `Style` типа `TBevelStyle` определяет стиль фаски и принимает следующие значения:

- `bsLowered` (фаска выглядит утопленной относительно поверхности размещения) — по умолчанию;
- `bsRaised` (фаска выглядит приподнятой относительно поверхности размещения).

Компонент *Image*

В VCL.NET компонент `Image` типа `TImage` служит для отображения изображения определенного графического формата. Он обычно помещается на поверхность формы и представляет собой невидимый контейнер для размещения реального изображения.

Компонент `Image` включает в себя класс `TPicture`, который, в свою очередь, имеет свойства и методы, используемые для работы с готовыми изображениями. Основным свойством этого компонента является свойство `Picture`. С его помощью можно, например, загрузить изображение.

Свойство `Picture` типа `TPicture` *определяет изображение*, размещаемое внутри компонента `Image`.

Объект типа `TPicture` является контейнером для графических объектов и может содержать растровое изображение форматов BMP, ICO или WMF. Для этого он включает в себя классы `TBitmap`, `TIcon` и `TMetaFile`. *Графическое изображение*, загруженное в объект типа `TPicture`, определяется свойством `Graphic` типа `TGraphic`. Это свойство можно использовать для доступа к изображениям любого из указанных типов, если он не известен. Если тип графики известен, то для операций с ней можно использовать свойства `Bitmap` типа `TBitmap`, `Icon` типа `TIcon` и `MetaFile` типа `TMetaFile`.

Свойства `Height` и `Width` типа `Integer` определяют соответственно *высоту* и *ширину* загруженного в объект типа `TPicture` изображения. Отметим, что значения этих свойств в общем случае не равны значениям одноименных свойств компонента `Image`, задающих размеры самого компонента `Image`: они совпадут только в случае, если изображение, загруженное в компонент `Image` с помощью свойства `Picture`, займет его полностью.

В объект типа `TPicture` изображение может загружаться из следующих источников:

- графический файл;
- компонент, содержащий изображение, например, `Image`;
- файл ресурса.

Изображение из графического файла можно загружать на этапе проектирования приложения (статически) и при его выполнении (динамически). Следует учитывать, что изображение, подключенное при проектировании приложения, соответственно увеличивает объем исполняемого файла программы. Чтобы избежать этого, рекомендуется загружать большие изображения динамически.

Для *загрузки изображения из файла* в классе `TPicture` имеется метод `LoadFromFile (const FileName: String)`, параметр `FileName` которого указывает графический файл-источник изображения. Данная процедура способна работать с файлами форматов BMP, WMF, ICO.

Например, инструкция

```
Image1.Picture.LoadFromFile('c:\picture\photo1.bmp');
```

загружает изображение из файла c:\picture\photo1.bmp в компонент `Image1`.

При *загрузке изображения* из содержащего его компонента для класса `TPicture` необходимо использовать свойство, указывающее тип графики в объекте-источнике.

Например, инструкция `Image2.Picture.Bitmap.Assign(Image1.Picture.Bitmap);` копирует изображение типа `TBitmap` из компонента `Image1` в компонент `Image2`.

Загрузка изображения из файла ресурсов для объекта `Bitmap` выполняется посредством метода `LoadFromResourceName (Instance: THandle; const ResName: String)`. Файл ресурса (RES) может быть подготовлен с помощью любого редак-

тора ресурсов. Перед использованием файл ресурса следует подключить к модулю директивой компилятора `$R`.

Пример загрузки изображения из файла ресурса:

```
{$R resource1.res}
...
Image1.Picture.BitMap.LoadFromResourceName(Instance, 'picture1');
```

В компонент `Image1` загружается изображение с именем `picture1`, содержащееся в файле ресурса resource1.res. Имя ресурса, в данном случае `picture1`, задается в редакторе ресурсов при создании изображения.

При необходимости *сохранение изображения*, содержащегося в компоненте, можно выполнить с помощью метода `SaveToFile` класса `TPicture`. Процедура `SaveToFile (const FileName: String)` сохраняет изображение, находящееся в контейнере `Picture`, на диске в файле с именем, заданным параметром `FileName`.

У компонента `Image` есть свойство `Canvas`, поэтому на его поверхности разрешается рисовать даже поверх уже находящегося на нем изображения. Так можно, например, оформить рамку вокруг рисунка или добавить к нему поясняющий текст.

Как отмечалось, размеры компонента `Image` и содержащегося в нем изображения (загруженного с помощью свойства `Picture`) в общем случае не совпадают. При этом возможна ситуация, когда изображение не помещается в области компонента `Image`. Для просмотра таких изображений можно использовать свойства `AutoSize` или `Stretch` компонента `Image`.

Свойство `AutoSize` типа `Boolean` управляет возможностью *автоматического приведения размеров* компонента `Image` к размерам содержащегося в нем изображения. Если свойство `AutoSize` установлено в значение `True`, то размеры элемента `Image` приводятся к размерам изображения, если в значение `False` (по умолчанию), то нет.

Свойство `Stretch` типа `Boolean` управляет возможностью *автоматического приведения размеров изображения* к размерам компонента `Image`, в котором оно содержится. Если свойство `Stretch` установлено в значение `True`, то размеры изображения приводятся к размерам компонента `Image`, если в значение `False` (по умолчанию), то размеры изображения не изменяются. Для значков это свойство не действует, они загружаются с исходными размерами.

При приведении размеров изображения к размерам компонента `Image` возможно нарушение пропорций изображения по высоте и ширине. Задав значение `True` для свойства `Proportional` типа `Boolean`, можно обеспечить сохранение пропорций изображения.

Если размеры изображения больше, чем размеры компонента `Image`, а свойства `AutoSize` и `Stretch` установлены в значение `False`, то часть изображения отсекает-

ся. Для обеспечения просмотра всего изображения, независимо от размеров области компонента, можно применять указанные ниже приемы.

- Установить значение `True` для свойства `Stretch`. При выводе изображения значительных размеров происходит сильная потеря качества рисунка из-за масштабирования. Поэтому такой способ используют, если качество отображения не играет существенной роли, например, в области предварительного просмотра открываемых графических файлов (как в компоненте `OpenPictureDialog1`).
- Установить значение `True` для свойства `AutoSize`. При этом потеря качества рисунка не происходит, т. к. размеры компонента `Image` подстраиваются под изображение, а не наоборот. Однако часто размеры компонента `Image` устанавливаются при разработке или имеют определенные пределы, связанные с дизайном формы, и изменять эти размеры нежелательно. В этом случае можно организовать прокрутку изображения. Для этого компонент `Image` помещают в контейнер (например, панель `Panel`), который ограничивает видимую область этого компонента. Перемещение видимой области осуществляется путем изменения значений свойств `Left` и `Top` компонента `Image`.

Свойство `Center` типа `Boolean` определяет, центрируется ли изображение внутри компонента `Image`. Если свойство установлено в значение `True`, то изображение центрируется, если свойство имеет значение `False` (по умолчанию), то изображение выравнивается по левому верхнему углу компонента `Image`.

Для работы с картинками в формате JPEG в VCL.NET предназначен класс `TJPEGImage`. Для использования этого класса и предоставляемых им возможностей в разделе `uses` следует подключить модуль `JPEG`.

Компонент *PaintBox*

Окно рисования (мольберт) представляет собой прямоугольную область, внутри которой можно выполнять операции рисования. Для работы с окном рисования в VCL.NET служит компонент `PaintBox` (расположен на странице **System** Палитры компонентов), основным свойством которого является свойство `Canvas` типа `TCanvas`. Окно рисования обычно используется, когда нужно ограничить поверхность рисования областью, размер которой меньше размера этой поверхности, а также для рисования на поверхности компонентов, не обладающих свойством `Canvas`.

Сам компонент `PaintBox` является невидимым, отображается только выводимое на нем изображение. В отличие от компонента `Image`, в компонент `PaintBox` нельзя загружать готовые изображения.

Компонент *ImageList*

Компонент `ImageList` (список графических изображений) предназначен для хранения набора графических изображений. Список представляет собой коллекцию

однотипных изображений одинакового размера, на каждое из которых можно ссылаться по индексу. Списки изображений используются для эффективного управления множествами значков или битовых массивов. Сам список является *невизуальным компонентом* и на экране не отображается, содержащиеся в нем изображения также являются невидимыми. Видимыми они становятся только при отображении их каким-либо способом на поверхности какого-либо визуального компонента. Списки изображений удобно использовать для запоминания различного числа рисунков, которые при необходимости можно быстро отобразить.

В VCL.NET список графических изображений представлен компонентом `ImageList` (расположен на странице **Win32** Палитры компонентов), производным от класса `TCustomImageList` и являющимся контейнером для хранения нескольких изображений одинакового типа и размера. Свойства компонента `ImageList` определяют его характеристики и характеристики содержащихся в нем изображений, а методы позволяют оперировать ими. Элементами списка могут быть изображения типа "значок" (ICO) или "битовый массив" (BMP).

Кроме того, в списке могут храниться *маски изображений*, которые определяют способ отображения при прорисовке. Размеры маски совпадают с размерами изображения. Нулевой бит маски указывает, что в этом месте изображения при его выводе на какой-либо поверхности будет нарисован бит с цветом фона. Ненулевой бит маски делает возможным вывод на прорисовываемой поверхности соответствующего ему бита изображения.

Элементы списка `ImageList` используются, например, для рисунков на кнопках `ToolButton` панели инструментов `ToolBar` или на ярлычках многостраничного блокнота `PageControl`. Эти компоненты имеют специальное свойство, его значение указывает имя списка графических изображений, из которого берутся рисунки. Таким свойством является, например, свойство `Images` типа `TCustomImageList`. Чтобы определить, какое именно изображение из заданного списка выводится в конкретном элементе, данный элемент имеет свойство, указывающее номер "своего" изображения. В частности, для кнопки `ToolButton` это свойство `ImageIndex` типа `Integer`.

Рисунок, выводимый на поверхности элемента, может зависеть от состояния этого элемента, например, вид глифа (от англ. glyph) на кнопке панели инструментов зависит от того, активна кнопка или заблокирована. Если для элемента используются несколько рисунков, то они берутся из различных списков, указанных соответствующими свойствами того компонента, которому эти элементы принадлежат. Так, для панели инструментов `ToolBar` свойства `Images`, `HotImages` и `DisableImages` могут задавать три различных контейнера для изображений, которые отображаются на кнопках `ToolButton` в зависимости от их состояния.

Для компонентов, у которых нет свойства, указывающего список графических изображений, но которые требуют отображения в своей области рисунков из списка, программист самостоятельно выполняет вывод графики с помощью класса `TCanvas` и списка `ImageList`.

Для работы с компонентом ImageList на этапе проектирования используется редактор (рис. 12.4), с помощью которого можно добавить в контейнер изображение или удалить его. Каждое изображение в компоненте ImageList имеет свой номер, отсчет начинается с нуля. Редактор позволяет перемещать отдельные изображения, изменяя их положения и, соответственно, их номера в контейнере. Кроме того, для отдельных изображений можно установить значения некоторых свойств, например, прозрачный (фоновый) цвет. Для каждого изображения может создаваться и запоминаться маска.

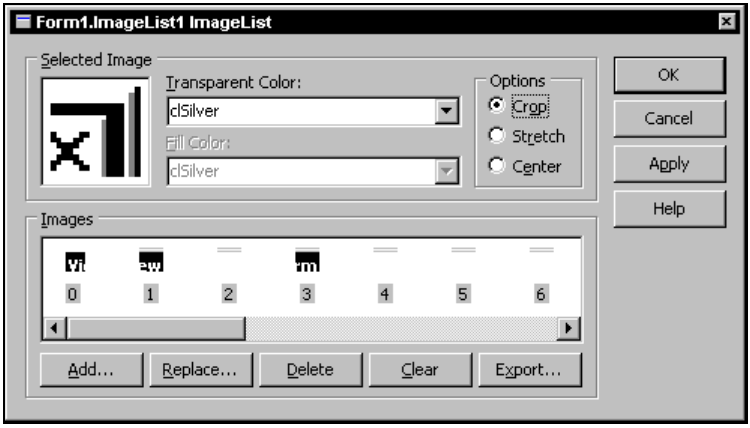


Рис. 12.4. Редактор компонента ImageList

Обычно при проектировании приложения в список графических изображений только загружается набор рисунков, а все остальные действия с этим списком реализуются при выполнении приложения.

Все хранимые в списке изображения имеют одинаковый размер. По умолчанию он составляет 16×16 пикселов и устанавливается при создании компонента ImageList. При необходимости можно динамически *установить новый размер*. Для этого используется конструктор CreateSize (AWidth, AHeight: Integer), в котором параметры AWidth и AHeight задают ширину и высоту изображения в пикселах.

Изменить размеры изображений в списке ImageList можно также, установив значения свойств Width и Height типа Integer, которые определяют ширину и высоту изображения и доступны для записи.

Замечание

При изменении значения любого из этих свойств все содержимое списка автоматически очищается. Поэтому значения этих свойств нужно устанавливать *до* загрузки изображений.

При использовании метода CreateSize подобного не происходит, и список ImageList сохраняет свое содержимое.

Для определения *числа изображений* в списке предназначено свойство `Count` типа `Integer`. При добавлении или удалении изображений значение этого свойства изменяется автоматически, также автоматически изменяются номера изображений в списке.

Загрузку изображений в контейнер можно выполнить не только с помощью редактора, но и динамически, в процессе выполнения программы. Для этого предназначены методы `Add`, `AddMasked`, `AddIcon`, `AddImages`, `Assign`, `Insert`, `InsertMasked`, `InsertIcon`, `Replace`, `ReplaceMasked` и `ReplaceIcon`.

Функция `Add (Image, Mask: TBitmap): Integer` *добавляет* в конец списка битовый массив и маску, заданные параметрами `Image` и `Mask` соответственно. В качестве результата возвращается значение индекса нового изображения в списке. Изображение и маска должны быть подготовлены до вызова данного метода.

Для *добавления изображений* удобно использовать функцию `AddMasked (Image: TBitmap; MaskColor: TColor): Integer`, которая формирует маску автоматически. Маска создается из пикселов изображения, указанного параметром `Image`, при этом учитывается цвет, заданный параметром `MaskColor`. В результате пикселы изображения, цвет которых совпадает с цветом `MaskColor`, рисуются прозрачными, а остальные выводятся своим цветом. Таким образом, параметр `MaskColor` задает фоновый цвет изображения. Если параметр `Image` содержит несколько изображений, они автоматически разделяются и добавляются к списку под разными номерами.

Приведем пример, в котором к списку `ImageList` добавляются новые рисунки.

```
procedure TForm1.Button1Click(Sender: TObject);
var bm: TBitmap;
begin
// Создание экземпляра объекта TBitmap
bm := TBitmap.Create;
// Загрузка рисунка из файла
bm.LoadFromFile('test.bmp');
// Очистка списка ImageList1
ImageList1.Clear;
// Добавление к списку ImageList1 трех рисунков.
// Фоновым является цвет левого верхнего пиксела изображения
ImageList1.AddMasked(bm, bm.Canvas.Pixels[0, 0]);
// Фоновым считается красный цвет
ImageList1.AddMasked(bm, clRed);
// Маска отсутствует
ImageList1.Add(bm, nil);
// Вывод на поверхности формы трех рисунков
ImageList1.Draw(Form1.Canvas, 30, 20, 0);
ImageList1.Draw(Form1.Canvas, 30, 50, 1);
ImageList1.Draw(Form1.Canvas, 30, 80, 2);
```

```
// Удаление экземпляра объекта TBitmap
bm.Free;
end;
```

Растровый рисунок из файла test.bmp загружается в переменную `bm` типа `TBitmap`, откуда три раза добавляется к списку `ImageList1`. Операторы добавления изображения к списку отличаются маской. В первом случае в качестве прозрачного цвета взят цвет левого верхнего пиксела картинки. Во втором случае прозрачным является красный цвет, а в последнем варианте маска отсутствует. После формирования списка графических изображений его содержимое отображается на поверхности формы.

Функция `AddIcon (Image: TIcon): Integer` служит для *добавления* в конец списка графических изображений значка, определенного параметром `Image`. Копирование маски зависит от значения свойства `Masked` типа `Boolean`. Если свойство `Masked` имеет значение `True`, то значок копируется вместе с маской, если `False`, то без маски.

Процедуры `AddImages (Value: TCustomImageList)` и `Assign (Source: TPersistent)` предназначены для *копирования* содержимого *одного списка графических изображений* в другой. Процедура `AddImages` *добавляет* в конец списка содержимое другого списка, указанного параметром `Value`. Процедура `Assign` *заменяет* старое содержимое списка новым, взятым из источника, заданного параметром `Source`.

Методы `Insert`, `InsertMasked` и `InsertIcon` отличаются от соответствующих методов `Add`, `AddMasked` и `AddIcon` тем, что позволяют *задавать позицию* списка, в которую вставляется указанное изображение.

Методы `Replace`, `ReplaceMasked` и `ReplaceIcon` *заменяют изображение* и *маску*, находящиеся на указанной позиции в списке, заданными изображением и маской.

Для *перемещения изображения* внутри списка служит метод `Move`. Процедура `Move (CurIndex, NewIndex: Integer)` перемещает изображение с позиции, указанной параметром `CurIndex`, на новое место, номер которого задан параметром `NewIndex`.

Для *удаления изображений* из компонента `ImageList` предназначены методы `Clear` и `Delete`. Процедура `Clear` *удаляет все содержимое* списка, а процедура `Delete (Index: Integer)` *удаляет изображение*, позиция которого в списке задана параметром `Index`. Напомним, что полная очистка списка изображений происходит также при изменении значений его свойств `Width` или `Height`.

Процедура `Draw (Canvas: TCanvas; X, Y, Index: Integer; Enabled: Boolean=True)` рисует на определяемой параметром `Canvas` поверхности *изображение*, номер которого задан параметром `Index`. Параметры `X` и `Y` указывают левый верхний угол, начиная с которого выводится рисунок. Параметр `Enabled` определяет доступность отображения, по умолчанию имеет значение `True` и обычно не указывается, при этом изображение доступно для обработки.

ГЛАВА 13

Работа с мультимедиа

На основе технологий мультимедиа можно добиться повышения выразительности и привлекательности программ. В составе средств мультимедиа наибольший интерес представляют средства, которые используют аудио- и видеовозможности компьютера.

Для применения средств мультимедиа в VCL.NET служат компоненты `Animate` и `MediaPlayer`.

Компонент `Animate`, расположенный на странице **Win32** Палитры компонентов, предназначен для проигрывания файлов формата AVI (Audio-Video Interleaved — "перемежаемые" изображение и звук). Компонент `MediaPlayer` расположен на странице **System** Палитры компонентов и представляет собой сложный *многофункциональный элемент*, который обеспечивает воспроизведение аудио- и видеофайлов, а также управление соответствующими устройствами. Во многом эти компоненты похожи друг на друга.

В простейших случаях для генерации звукового сигнала можно использовать процедуру `Beep` модуля `SysUtils`. Эта процедура вызывает одноименную API-функцию `Beep`, издающую стандартный системный звук с помощью встроенного динамика.

Рассмотрим следующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
try
   Edit1.Text := IntToStr(StrToInt(Edit1.Text) + 1);
   except
   Beep;
   MessageDlg('Ошибка в поле Edit1!', mtError, [mbOK], 0);
   end;
end;
```

Редактор `Edit1` содержит целое число. При нажатии кнопки `Button1` это число увеличивается на единицу. В случае возникновения какого-либо исключения, например ошибки преобразования, выдается звуковой сигнал и выводится сообщение.

Для получения звука можно использовать также API-функцию `MessageBeep (uType: UINT): Boolean`, генерирующую стандартный системный звук, тип которого указан параметром `uType`. Параметр `uType` задается в виде шестнадцатеричного числа или с помощью именованных констант: например, шестнадцатеричное число `$FFFFFFFF` соответствует стандартному системному звуку, а именованная константа `MB_OK` — системному звуку по умолчанию.

При успешном вызове функции `MessageBeep` воспроизведение звука осуществляется асинхронно (параллельно) с дальнейшим выполнением приложения, а в качестве результата возвращается значение `True`.

Выдаваемые звуковые сигналы зависят от настроек, выполненных на панели управления Windows. В частности, если невозможно воспроизвести звук заданного типа, то воспроизводится системный звук по умолчанию.

Воспроизведение видеоклипов

Видеоклип представляет собой файл в формате AVI. Этот файл содержит последовательность отдельных изображений (кадров), при отображении которых создается эффект движения изображения. Наряду с изображением, AVI-файлы могут содержать звук. Для воспроизведения видеоклипов подходит любой из компонентов `Animate` или `MediaPlayer`.

Компонент `Animate` позволяет проигрывать AVI-файлы, а также отображать стандартную анимацию, используемую Windows. Однако AVI-файлы, воспроизводимые этим компонентом, имеют определенные ограничения:

- они не должны содержать звука;
- информация в них должна быть не сжатой;
- размер файла не должен превышать 64 Кбайт.

Для задания воспроизводимого видеоклипа используются свойства `FileName` типа `TFileName` и `CommonAVI` типа `TCommonAVI`. Одновременно можно применять одно из этих свойств: так, если в качестве значения свойства `FileName` указывается AVI-файл, существующий на диске, то свойство `CommonAVI` автоматически устанавливается в значение `aviNone`. Свойство `CommonAVI` позволяет выбрать один из стандартных клипов Windows, обеспечиваемых в Shell32.dll, и принимает следующие значения:

- `aviNone` (отсутствие стандартной анимации);
- `aviCopyFile` (копирование файла);

- `aviCopyFiles` (копирование файлов);
- `aviDeleteFile` (удаление файла);
- `aviEmptyRecycle` (очистка Корзины);
- `aviFindComputer` (поиск компьютера);
- `aviFindFile` (поиск файла);
- `aviFindFolder` (поиск папки);
- `aviRecycleFile` (перемещение файла в корзину).

При установке свойству `CommonAVI` отличного от `aviNone` значения свойство `FileName` автоматически сбрасывается, принимая значение пустой строки.

Для задания видеоклипа также можно использовать свойства `ResHandle` типа `THandle` и `ResID` типа `Integer`, представляющие собой альтернативы свойствам `CommonAVI` и `FileName`. Значение свойства `ResHandle` *задает ссылку на модуль*, который содержит изображение в виде ресурса, а значение свойства `ResID` *указывает номер ресурса* в этом модуле. В случае успешной загрузки видеоклипа из ресурса свойству `CommonAVI` автоматически устанавливается значение `aviNone`, а значение свойства `FileName` сбрасывается в пустую строку.

После выбора видеоклипа свойства `FrameCount`, `FrameHeight` и `FrameWidth` типа `Integer` определяют *число кадров*, а также *высоту* и *ширину кадров* (в пикселах) соответственно. Все эти свойства доступны только для чтения во время выполнения программы.

По умолчанию размеры компонента `Animate` автоматически подстраиваются под размеры кадров видеоклипа, это определяет значение `True` свойства `AutoSize` типа `Boolean`. Если этому свойству установить значение `False`, то компонент `Animate` свои размеры не изменяет, при этом возможно отсечение части кадра изображения, если его размеры превышают размеры компонента `Animate`.

Воспроизведение видеоклипа начинается при установке свойству `Active` типа `Boolean` значения `True`. *Начальный* и *конечный кадры* задают диапазон воспроизведения и определяются, соответственно, значениями свойств `StartFrame` и `StopFrame` типа `SmallInt`. По умолчанию свойство `StartFrame` указывает на первый кадр анимации и имеет значение `1`, а свойство `StartFrame` — на последний кадр, и его значение равно значению свойства `FrameCount`.

Свойство `Repetitions` типа `Integer` определяет *число повторений* воспроизведения видеоклипа. По умолчанию оно имеет значение `0`, и видеоклип проигрывается неограниченное число раз до тех пор, пока процесс воспроизведения не будет остановлен.

Для *принудительной* (до истечения заданного числа повторений) *остановки воспроизведения* видеоклипа свойству `Active` следует установить значение `False`.

Свойство `Center` типа `Boolean` задает, будет ли изображение выводиться в центре компонента `Animate` или в его левом верхнем углу. По умолчанию это свойство

имеет значение `True`, и видеоклип проигрывается в центре области компонента `Animate`.

Для запуска и остановки воспроизведения клипов можно использовать также методы `Play`, `Stop` и `Reset`. Процедура `Play (FromFrame, ToFrame: Word; Count: Integer)` *проигрывает видеоклип*, начиная с кадра, заданного параметром `FromFrame`, и заканчивая кадром, заданным параметром `ToFrame`. Параметр `Count` определяет число повторений. Таким образом, эта процедура позволяет одновременно управлять свойствами `StartFrame`, `StopFrame` и `Repetitions`, задавая для них требуемые при воспроизведении значения, а также устанавливает свойству `Active` значение `True`.

Процедура `Stop` *прерывает воспроизведение* видеоклипа и устанавливает свойству `Active` значение `False`. Процедура `Reset`, кроме того, дополнительно сбрасывает свойства `StartFrame` и `StopFrame`, устанавливая для них значения по умолчанию.

Свойство `Open` типа `Boolean` доступно при выполнении программы и позволяет определить, *готов ли компонент* `Animate` к воспроизведению. Если выбор и загрузка видеоклипа проходят успешно, свойство `Open` автоматически принимает значение `True`, и компонент `Animate` можно открыть — проиграть анимацию. Если загрузить видеоклип не удается, то это свойство получает значение `False`. При необходимости программист может сам устанавливать свойству `Open` значение `False`, отключая этим компонент `Animate`.

Рассмотрим пример анимации часов на форме приложения с помощью просмотра файла clock.avi (рис. 13.1).

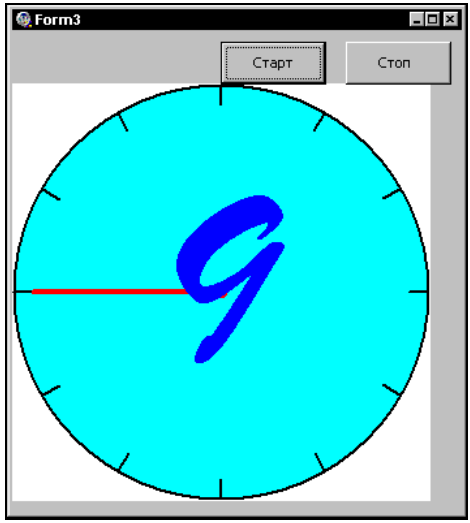


Рис. 13.1. Анимация часов

Управление запуском анимации часов в приложении выполняется с помощью двух обработчиков событий `Click` для кнопок:

```
procedure TForm3.Button1Click(Sender: TObject);
begin
    Animate1.FileName:='c:\winnt\clock.avi';
    Animate1.Active :=True;
end;

procedure TForm3.Button2Click(Sender: TObject);
begin
    Animate1.Active :=False;
end;
```

В приведенном примере звук отсутствует.

Как говорилось, компонент `Animate` обеспечивает воспроизведение только простых AVI-файлов. Для проигрывания больших файлов предназначен компонент `MediaPlayer`, предоставляющий гораздо более богатые мультимедийные возможности.

Часто компонент `Animate` используется при *создании панелей инструментов* `ToolBar` или `CoolBar` для *добавления в них анимационных значков*, которые оживляют форму и служат для индикации того, что программа выполняет ту или иную обработку данных. Воспроизведение изображения при этом осуществляется, например, при нажатии кнопки панели инструментов или по истечении заданного интервала времени.

Управление мультимедийными устройствами

Многие мультимедийные устройства имеют так называемый MCI-интерфейс (Media Control Interface — интерфейс управления мультимедиа). С помощью MCI-интерфейса осуществляется управление объектами типа аудио- и видеофайлов, а также устройствами мультимедиа, такими как звуковые карты, карты обработки видеосигналов, видеомагнитофоны, лазерные проигрыватели. В приложениях VCL.NET работу с такими устройствами осуществляет компонент `MediaPlayer` (мультимедийный проигрыватель), который обеспечивает стыковку приложения с драйверами MCI.

Мультимедийный проигрыватель `MediaPlayer` является многофункциональным элементом управления и предоставляет программисту большой набор свойств и методов, позволяющих манипулировать файлами и устройствами мультимедиа, поддерживать воспроизведение, запись и перемещение между отдельными фонограммами (дорожками, записями), а также идентифицировать подключенные устройства.

Визуально компонент `MediaPlayer` представляет собой набор кнопок (рис. 13.2), с помощью которых формируется панель управления. Эту панель можно использовать для управления различными аппаратными и программными средствами мультимедиа.

Рис. 13.2. Вид компонента `MediaPlayer`

Компонент `MediaPlayer` содержит следующие кнопки (рис. 13.2, слева направо):

- **Play** (воспроизведение);
- **Pause** (пауза);
- **Stop** (остановка);
- **Next** (переход к следующей фонограмме (дорожке)); если фонограмма только одна, выполняется переход в ее конец;
- **Prev** (переход к предыдущей фонограмме); если фонограмма только одна, выполняется переход в ее начало;
- **Step** (переход на несколько кадров вперед);
- **Back** (возврат на несколько кадров назад);
- **Record** (включение режима записи);
- **Eject** (извлечение носителя).

Видимостью и *доступностью* кнопок мультимедийного проигрывателя можно управлять с помощью свойств `VisibleButtons`, `EnabledButtons` и `ColoredButtons`.

Свойство `VisibleButtons` типа `TButtonSet` определяет, какие кнопки в компоненте `MediaPlayer` являются *видимыми*. По умолчанию видимы все кнопки, что не всегда удобно. Например, при воспроизведении звуковых файлов формата WAV кнопки **Step**, **Back** и **Eject** не нужны, и лучше сделать их невидимыми. Тип `TButtonSet` содержит значения, соответствующие кнопкам проигрывателя, и описан так:

```
type TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep,
                   btBack, btRecord, btEject);
  TButtonSet = set of TMPBtnType;
```

Обычно набор видимых кнопок мультимедийного проигрывателя устанавливается через Инспектор объектов, но можно задать его и при выполнении программы. Например, если требуется отображать только кнопки воспроизведения, паузы и остановки, то достаточно следующей инструкции:

```
MediaPlayer1.VisibleButtons := [btPlay,btPause,btStop];
```

Свойство `EnabledButtons` типа `TButtonSet` определяет, какие кнопки *доступны* в элементе мультимедиа. Доступная кнопка выделяется цветом и может быть нажата. *Недоступная* (запрещенная) кнопка имеет светло-серый цвет и не реагирует на нажатие. По умолчанию доступны все кнопки, однако мультимедиа-проигрыватель в зависимости от ситуации может сам *управлять доступностью* своих кнопок, что определяется свойством `AutoEnable` типа `Boolean`. Если это свойство имеет значение `True` (по умолчанию), то проигрыватель автоматически изменяет доступность отдельных кнопок в зависимости от устройства, которым он манипулирует, и режима, в котором он находится. Например, если воспроизводится звуковой файл, то блокируется кнопка **Eject**. Таким образом, если свойству `AutoEnable` установлено значение `True`, то программист может управлять доступностью только тех кнопок проигрывателя, которые не заблокированы автоматически самим проигрывателем.

При выполнении программы пользователь нажимает кнопки с помощью мыши или клавиатуры. В случае использования клавиатуры нажатие выбранной кнопки выполняется клавишей <Space>, а перемещение между кнопками проигрывателя производится с помощью клавиш <←> и <→>. При нажатии кнопки вызывается соответствующий метод, выполняющий требуемые действия, например, метод `Stop` осуществляет остановку воспроизведения.

Название и назначение большинства таких методов (`Play`, `Pause`, `Stop`, `Next`, `Step`, `Back` и `Eject`) совпадают с названием и назначением вызывающих их кнопок. Исключениями являются метод `StartRecording`, выполняемый при нажатии кнопки **Record**, и метод `Previous`, которому соответствует кнопка **Prev**. Обычно при управлении компонентом `MediaPlayer` программист вызывает эти методы самостоятельно.

В процессе работы мультимедийный проигрыватель связан с конкретным файлом на внешнем носителе. Этот файл открыт для воспроизведения и/или записи, а *имя файла* определяет свойство `FileName` типа `String`.

В качестве примера рассмотрим процедуру, в которой нажатие кнопки сопровождается звуковым эффектом в виде щелчка:

```
procedure TForm3.Button1Click(Sender: TObject);
begin
   MediaPlayer1.Visible    := False;
        MediaPlayer1.DeviceType := dtAutoSelect;
        MediaPlayer1.FileName   := 'c:\winnt\media\start.wav';
        try
           if not MediaPlayer1.AutoOpen then MediaPlayer1.Open;
           MediaPlayer1.Play;
           except
           exit;
      end;
end;
```

При обработке события `Button1Click` воспроизводится щелчок с помощью звукового файла start.wav. При возникновении исключения, связанного, например, с ошибкой чтения файла, звук отсутствует, и предупреждающее сообщение не выдается.

Можно для нескольких кнопок задать общий обработчик события с аналогичным кодом. В этом случае действия пользователя по работе с кнопками будут иметь звуковое сопровождение.

В каждый момент времени мультимедийный проигрыватель может управлять только одним *устройством*, задаваемым в свойстве `DeviceType` типа `TMPDeviceTypes`. Это свойство принимает следующие значения:

- `dtAutoSelect` (автоматическое распознавание типа устройства);
- `dtAVIVideo` (AVI-файл);
- `dtCDAudio` (аудио-CD);
- `dtDAT` (цифровая аудиолента);
- `dtDigitalVideo` (цифровое видео); допускаются файлы AVI, MPG, MOV или MM-фильм;
- `dtMMMovie` (MM-фильм);
- `dtOther` (другое устройство);
- `dtOverlay` (аналоговое видео);
- `dtScanner` (сканер);
- `dtSequencer` (MIDI-файл);
- `dtVCR` (видеокассета);
- `dtVideodisc` (видео-CD);
- `dtWaveAudio` (WAV-файл).

По умолчанию установлено значение `dtAutoSelect`, что означает автоматическое распознавание типа открытого устройства в зависимости от расширения имени файла, указанного в свойстве `FileName`. Напомним, что расширения имен файлов связаны с конкретными устройствами и средствами Windows.

Перед использованием устройства его следует открыть, поскольку большинство методов, например, `Play` и `StartRecording`, можно вызывать только после открытия устройства. Открытие устройства выполняется вызовом метода `Open`. Если необходимо автоматическое открытие устройства, то свойству `AutoOpen` типа `Boolean` следует установить значение `True`. По умолчанию это свойство имеет значение `False`, и при создании формы устройство, связанное с компонентом `MediaPlayer`, автоматически не открывается.

После того как устройство открыто, свойство `DeviceID` типа `Word` проигрывателя определяет идентификатор этого устройства. Если открытых устройств нет, то свойство `DeviceID` имеет значение ноль.

Например, в приводимой ниже процедуре открывается проигрыватель аудио-CD:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
MediaPlayer1.DeviceType := dtCDAudio;
MediaPlayer1.Open;
end;
```

Когда использование мультимедийного устройства прекращается, его нужно закрыть, вызвав метод `Close`. Этот метод вызывается автоматически при завершении работы с приложением и при удалении формы, в которой находится компонент `MediaPlayer`.

Свойство `Capabilities` типа `TMPDevCapsSet` позволяет определить возможности выбранного и открытого мультимедийного устройства. Это свойство может принимать набор следующих значений, устанавливающих доступность соответствующих операций:

- `mpCanEject` (извлечение носителя);
- `mpCanPlay` (воспроизведение);
- `mpCanRecord` (запись на носитель);
- `mpCanStep` (переход вперед или назад на определенное число кадров);
- `mpUsesWindow` (использование окна для вывода изображения).

После открытия устройства с помощью свойства `Tracks` типа `Longint` можно получить информацию о *числе фонограмм* (дорожек); если устройство не поддерживает дорожки, то значение этого свойства не определено. Свойство `TrackLength[TrackNum: Integer]` типа `Longint` содержит *длину фонограммы* с индексом `TrackNum` (отсчет начинается с единицы). Длина дорожки указывается в формате времени, который определен свойством `TimeFormat`.

Свойство `TimeFormat` типа `TMPTimeFormats` задает *формат значений* свойств, связанных со временем. Оно влияет на способ интерпретации и отображения значений таких свойств, как `TrackLength`, `Length`, `StartPos`, `EndPos`, `Position`. Каждое устройство поддерживает определенные для него форматы времени. Свойство `TimeFormat` принимает следующие значения:

- `tfMilliseconds` (целое четырехбайтовое число, определяет количество миллисекунд);
- `tfHMS` (числа (определяют часы, минуты и секунды), размещенные побайтно, начиная с младшего байта, в четырехбайтовом целом; старший байт не учитывается);
- `tfMSF` (числа (определяют минуты, секунды и количество кадров), размещенные побайтно, начиная с младшего байта, в четырехбайтовом целом; старший байт не используется);

- `tfFrames` (целое четырехбайтовое число, содержит количество кадров);
- `tfSMPTE24`, `tfSMPTE25`, `tfSMPTE30` и `tfSMPTE30Drop` — каждое из первых трех значений определяет часы, минуты, секунды и число блоков по 24, 25 и 30 кадров в секунду соответственно; последнее значение определяет часы, минуты, секунды и число пропущенных блоков по 30 кадров в секунду;
- `tfBytes` (четырехбайтовое целое, содержит количество байтов);
- `tfSamples` (четырехбайтовое целое, содержит количество условных блоков информации);
- `tfTMSF` (числа (определяют дорожки, минуты, секунды и количество кадров), размещенные побайтно, начиная с младшего байта, в четырехбайтовом целом).

Свойство `Position` типа `Longint` доступно для чтения при выполнении программы и указывает *текущую позицию* открытого устройства в установленном формате времени. При первоначальном открытии носителя значение свойства `Position` устанавливается в начало или в позицию, определенную свойством `StartPos`.

Свойства `StartPos` и `EndPos` типа `Longint` определяют, соответственно, *начальную* и *конечную позиции*, в пределах которых осуществляется воспроизведение. По умолчанию значение свойства `StartPos` устанавливается в начальную позицию носителя, а значение свойства `EndPos` — в конечную. Значения указываются в текущем формате времени. Оба свойства доступны для записи во время выполнения программы. Установки, сделанные для свойств `StartPos` и `EndPos`, действуют только на следующий вызов методов `Play` или `StartRecording`, после чего для них снова устанавливаются значения по умолчанию. При необходимости повторного вызова желаемые значения начала и конца воспроизведения должны устанавливаться заново.

В следующей процедуре проигрывается звуковой файл test.wav:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
with MediaPlayer1 do begin
     FileName := 'test.wav';
     Open;
     TimeFormat := tfMilliseconds;
     StartPos   := 10000;
     EndPos     := 25000;
     Play;
     end;
end;
```

Воспроизведение файла осуществляется с 10-й по 25-ю секунду записи. Время отсчитывается в миллисекундах.

При воспроизведении компонентом `MediaPlayer` видеоизображений свойство `Display` типа `TWinControl` задает *имя оконного элемента управления*, в котором

осуществляется отображение. По умолчанию свойство `Display` имеет значение `nil`, и для отображения создается собственное окно. Программист может задать для вывода свое окно или какой-либо компонент, например, панель. Свойство `DisplayRect` типа `TRect` определяет *прямоугольную область*, используемую для вывода изображения, значение этого свойства следует устанавливать после открытия проигрывателя. Оба свойства доступны во время выполнения приложения. Вывод в окно поддерживают устройства таких типов, как `dtAVIVideo`, `dtDigitalVideo`, `dtOverlay`, `dtVCR`, `dtVideodisc`.

Пример отображения видеофайла:

```
procedure TForm1. FormCreate(Sender: TObject);
begin
With MediaPlayer1 do begin
     DeviceType := dtAVIVideo;
       FileName := ExtractFilePath(Application.ExeName) + 'move.avi';
     Open;
     end;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
With MediaPlayer1 do begin
     Display := Panel1;
     Play;
     end;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
With MediaPlayer1 do begin
     Display := nil;
     Play;
     end;
end;
```

При нажатии кнопки `Button1` видеофайл move.avi отображается на поверхности компонента `Panel1`, а при нажатии кнопки `Button2` для вывода изображения используется отдельное окно. Загрузка файла и установка типа устройства выполняются при создании формы.

Текущий *режим* мультимедийного проигрывателя указывает свойство `Mode` типа `TMPModes`, принимающее следующие значения:

- `mpNotReady` (устройство не готово);
- `mpStopped` (текущая операция завершена);
- `mpPlaying` (выполняется воспроизведение);
- `mpRecording` (выполняется запись);

- `mpSeeking` (выполняется поиск файла на носителе);
- `mpPaused` (операция приостановлена);
- `mpOpen` (устройство открыто).

Свойство `Error` типа `Longint` содержит *код ошибки*, возвращенный при выполнении устройством последней операции. Если последняя операция завершилась успешно, то в `Error` записывается `0`. Получить *описание ошибки* можно с помощью свойства `ErrorMessage` типа `String`, которое содержит строку, поясняющую последнюю возникшую ошибку.

Рассмотрим пример:

```
procedure TForm1.Button6Click(Sender: TObject);
begin
MediaPlayer1.Next;
if MediaPlayer1.Error = 0
   then MediaPlayer1.Play
   else MessageDlg(MediaPlayer1.ErrorMessage, mtError, [mbOK], 0);
end;
```

В компоненте `MediaPlayer1` выполняется переход на следующую фонограмму. Если операция выполнена успешно и код ошибки равен нулю, то фонограмма воспроизводится. В противном случае выдается сообщение об ошибке.

Для обработки ошибок, возникающих при работе с компонентом `MediaPlayer`, также могут использоваться рассмотренные ранее конструкции `try .. except` и `try .. fynally`.

При перемещении носителя с помощью методов `Step` и `Back` свойство `Frames` типа `Longint` определяет *количество кадров*, на которое изменяется текущая позиция. Свойство доступно во время выполнения программы, значение по умолчанию составляет 10% от длины текущего носителя.

Свойство `AutoRewind` типа `Boolean` определяет, выполняет ли мультимедийное устройство *перемотку* перед воспроизведением или записью. Если свойство `AutoRewind` имеет значение `True` (по умолчанию), то текущая позиция устройства перемещается в начало. В противном случае воспроизведение или запись начинается с текущей позиции. Свойство `AutoRewind` не действует, если в устройстве используются дорожки или если программист присвоил какие-либо значения свойствам `StartPos` и `EndPos`.

Если открытое устройство поддерживает режим записи, то его носитель можно с помощью метода `Save` *сохранить в файле*, заданном свойством `FileName`. Для устройств типа аудио-CD метод игнорируется.

После своего вызова каждый метод компонента `MediaPlayer` может *возвращать управление приложению* различными способами, определяемыми названием метода и значением свойства `Wait` типа `Boolean`, доступным во время выполнения. Если свойство `Wait` имеет значение `False`, то приложение продолжает свою рабо-

ту обычным порядком, а если значение True, то приложение ожидает завершения выполнения метода. Свойство Wait воздействует только на следующий вызов какого-либо метода и после вызова этого метода снова принимает значение по умолчанию, поэтому при необходимости его следует переустановить. По умолчанию для методов Play и StartRecording свойство Wait имеет значение False, а для остальных методов — значение True.

Так, в процедуре

```
procedure TForm1.Button5Click(Sender: TObject);
begin
MediaPlayer1.Wait := True;
MediaPlayer1.Play;
end;
```

компонент MediaPlayer1 переводится в режим воспроизведения, при этом работа с приложением блокируется до окончания воспроизведения.

С нажатием кнопок мультимедийного проигрывателя связано несколько событий, из которых наиболее часто используются события OnClick и OnNotify.

Событие OnClick типа EMPNotify возникает при нажатии какой-либо кнопки. Тип EMPNotify описан так:

```
type EMPNotify = procedure (Sender: TObject; Button: TMPBtnType;
     var DoDefault: Boolean) of object;
```

Параметр Button содержит информацию *о нажатой кнопке* и принимает следующие значения: btPlay, btPause, btStop, btNext, btPrev, btStep, btBack, btRecord, btEject. Перечисленные значения соответствуют приведенным ранее названиям кнопок проигрывателя, к которым добавлен префикс bt. Анализ этого параметра может понадобиться в случае, когда требуется определить, какую кнопку проигрывателя нажал пользователь.

Значение логического параметра DoDefault для нажатой кнопки определяет, *нужно ли выполнять стандартные действия*. Если параметр имеет значение True (по умолчанию), то для всех кнопок выполняются стандартные действия, например, при нажатии кнопки **Stop** вызывается соответствующий метод Stop компонента MediaPlayer. Если программист обрабатывает нажатия кнопок самостоятельно, то необходимый код размещается в обработчике события OnClick, а параметру DoDefault присваивается значение False.

Событие OnNotify типа TNotifyEvent возникает *при завершении* какого-либо из методов проигрывателя. Генерация этого события зависит от названия метода и значения свойства Notify типа Boolean. Событие OnNotify генерируется для каждого вызова методов Play и StartRecording, если перед их вызовом свойство Notify не было установлено в значение False. Другие методы проигрывателя, например, Stop, Close или Next не вызывают это событие, если свойство Notify

не установлено в значение `True`. После обработки события `OnNotify` свойство `Notify` необходимо снова установить в значение `True` для обеспечения генерации следующего такого события.

Приведем пример обработчика, в котором осуществляется запуск проигрывателя файлов мультимедиа с возможностью их выбора с помощью стандартного диалога открытия файла (рис. 13.3).

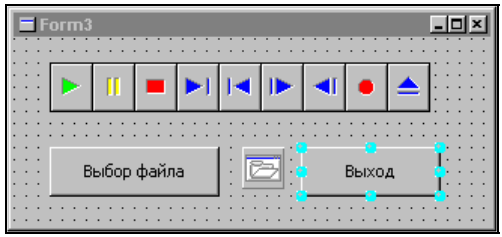


Рис. 13.3. Проигрыватель файлов мультимедиа при разработке

Обработчик события нажатия кнопки `Button1` с заголовком **Выбор файла** содержит следующий код:

```
procedure TForm3.Button1Click(Sender: TObject);
begin
   OpenDialog1.Filter := 'Файлы мультимедиа (*.wav; *.mid; *.avi) |
                                                    *.wav;*.mid;*.avi';
   MediaPlayer1.DeviceType := dtAutoSelect;
   MediaPlayer1.AutoEnable := True;
   MediaPlayer1.AutoOpen   := False;
   if OpenDialog1.Execute then begin
      MediaPlayer1.FileName := OpenDialog1.FileName;
      MediaPlayer1.Open;
      end;
end;
```

Здесь осуществляется автоматическое определение типа проигрывающего устройства, устанавливается возможность доступа для управления проигрывателем, запрещается автоматическое открытие проигрывателя. Выбор файла для воспроизведения осуществляется в диалоговом окне (компонент `OpenDialog1`) открытия файла. В случае подтверждения выбора указанный файл загружается в проигрыватель. Пользователь управляет проигрывателем с помощью его кнопок.

ЧАСТЬ IV

Работа с базами данных .NET

ГЛАВА 14

Введение в базы данных

Основные понятия

Для успешного функционирования различным организациям требуется развитая информационная система, реализующая автоматизированный процесс сбора, манипулирования и обработки данных.

Банки данных

Современной формой информационных систем являются *банки данных*, имеющие в своем составе:

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

База данных (БД) обеспечивает хранение информации, а также удобный и быстрый доступ к данным. Она представляет собой совокупность данных различного характера, организованных по определенным правилам. Информация в БД должна быть:

- непротиворечивой;
- неизбыточной;
- целостной.

Система управления базой данных (СУБД) — это совокупность языковых и программных средств, предназначенных для создания, ведения и использования БД. По характеру применения СУБД разделяют на персональные и многопользовательские.

Персональные СУБД обеспечивают возможность создания локальных БД, работающих на одном компьютере. К персональным СУБД относятся Paradox, dBase, FoxPro, Access и др.

Многопользовательские СУБД позволяют создавать информационные системы, функционирующие в архитектуре "клиент-сервер". Наиболее известными многопользовательскими СУБД являются Oracle, Informix, SyBase, Microsoft SQL Server, InterBase.

В состав языковых средств современных СУБД входят:

- язык описания данных, предназначенный для описания логической структуры данных;
- язык манипулирования данными, обеспечивающий выполнение основных операций над данными — ввод, модификацию и выборку;
- язык структурированных запросов (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.

Прикладные программы, или *приложения*, служат для обработки данных, содержащихся в БД. Пользователь осуществляет управление БД и работу с ее данными именно с помощью приложений, которые также называют *приложениями БД*.

Иногда термин "база данных" трактуют в более широком смысле и обозначают им не только саму БД, но и приложения, обрабатывающие ее данные.

Замечание

Delphi .NET 2006 обладает вполне развитыми возможностями СУБД. Предоставляемые ею средства обеспечивают создание и ведение локальных и клиент-серверных БД, а также разработку приложений для работы практически с любыми БД.

Модели данных

База данных содержит данные, используемые какой-либо прикладной информационной системой (например, системами "Сирена" или "Экспресс" продажи авиа- и железнодорожных билетов).

В зависимости от вида организации данных различают следующие основные модели представления данных в базе:

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

Большинство современных БД для персональных компьютеров являются реляционными. При последующем изложении материала речь пойдет именно о реляционных БД.

Реляционная модель, предложенная Эдгаром Коддом, получила название от английского термина *relation* (отношение). Реляционная БД представляет собой совокупность таблиц, *связанных отношениями*. Достоинствами реляционной модели данных являются простота, гибкость структуры, удобство реализации на компьютере, наличие теоретического описания.

Базы данных и приложения

По взаимному расположению приложения и БД можно выделить:

- ❒ локальные БД;
- ❒ удаленные БД.

Для выполнения операций с локальными БД разрабатываются и используются так называемые *локальные приложения*, а для операций с удаленными БД — *клиент-серверные приложения*.

Расположение БД в значительной степени влияет на разработку приложения, обрабатывающего содержащиеся в этой базе данные.

Так, различают следующие виды приложений:

- ❒ приложения, использующие локальные базы данных, называют *одноуровневыми* (однозвенными) приложениями, поскольку приложение и базы данных образуют единую файловую систему;
- ❒ приложения, использующие удаленные базы данных, разделяют на двухуровневые (двухзвенные) и многоуровневые (многозвенные). *Двухуровневые* приложения содержат клиентскую и серверную части;
- ❒ *многоуровневые* (обычно трехуровневые) приложения кроме клиентской и серверной частей имеют дополнительные части. К примеру, в трехуровневых приложениях имеются клиентская часть, сервер приложений и сервер базы данных.

Одно- и двухуровневые приложения Windows Forms .NET доступ к базам данных осуществляют с помощью одного основного механизма — ADO.NET совместно с провайдерами данных BDP.NET. ADO.NET представляет собой объектную модель, которая использует стандарт XML для передачи данных. В нем реализована идея использования *отсоединенных* наборов данных.

Трехуровневые приложения Windows Forms .NET можно создавать с помощью BDP.NET.

Одно- и двухуровневые приложения VCL.NET могут осуществлять доступ к локальным и удаленным БД с использованием следующих механизмов:

- BDE.NET (Borland Database Engine — процессор баз данных фирмы Borland), предоставляющий развитый интерфейс API для взаимодействия с базами данных;
- dbGo.NET на основе ADO (ActiveX Data Objects — объекты данных ActiveX) осуществляет доступ к информации с помощью OLE DB (Object Linking and Embedding Data Base — связывание и внедрение объектов баз данных). При этом используются компоненты, расположенные на странице dbGo;
- dbExpress.NET обеспечивает быстрый доступ к информации в базах данных с помощью набора драйверов;
- InterBase.NET реализует непосредственный доступ к базам данных InterBase.

Выбор варианта технологии доступа к информации в базах данных, кроме прочих соображений, определяется с учетом удобства подготовки разработанного приложения к распространению, а также дополнительного расхода памяти.

Трехуровневые приложения VCL.NET можно создавать с помощью механизма DataSnap. Используемые при создании трехуровневых (многоуровневых) приложений баз данных компоненты расположены на страницах **DataSnap** и **Data Access** Палитры компонентов.

Варианты архитектуры

Здесь мы рассмотрим коротко различные варианты архитектуры информационной системы.

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

Удаленная БД размещается на компьютере-сервере сети, а приложение, осуществляющее работу с этой БД, находится на компьютере пользователя. В этом случае мы имеем дело с архитектурой "клиент-сервер", когда информационная система делится на неоднородные части — сервер и клиент БД. В связи с тем, что компьютер-сервер отделен от клиента, его также называют *удаленным сервером*.

Клиент — это приложение пользователя. Для получения данных клиент формирует и отсылает запрос удаленному серверу, на котором размещена БД. Запрос формулируется на языке SQL, который является стандартным средством доступа к серверу при использовании реляционных моделей данных. После получения

запроса удаленный сервер направляет его программе SQL Server (серверу баз данных) — специальной программе, управляющей удаленной БД и обеспечивающей выполнение запроса и выдачу его результатов клиенту.

В архитектуре "клиент-сервер" клиент посылает запрос на предоставление данных и получает только те данные, которые действительно были затребованы. Вся обработка запроса выполняется на удаленном сервере. Такая архитектура обладает следующими достоинствами:

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

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

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

Реляционные базы данных

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

Таблицы баз данных

Таблицы, образующие БД, находятся в каталоге (папке) на жестком диске. Таблицы хранятся в файлах и похожи на отдельные документы или электронные таблицы (например, табличного процессора Microsoft Excel), их можно перемещать и копировать обычным способом, скажем, с помощью Проводника Windows. Однако в отличие от документов, таблицы БД поддерживают *многопользовательский* режим доступа, это означает, что их могут одновременно использовать несколько приложений.

Для одной таблицы создается несколько файлов, содержащих данные, индексы, ключи и т. п. Главным из них является файл с данными, имя этого файла совпадает с именем таблицы, которое задается при ее создании. В некотором смысле понятия таблицы и ее главного файла являются синонимами, при выборе таблицы выбирается именно ее главный файл: для таблицы dBase это файл с расширением dbf, а для таблицы Paradox — файл с расширением db. Имена остальных файлов таблицы назначаются автоматически — все файлы имеют одинаковые имена, совпадающие с именами таблиц, и разные расширения, указывающие на содержимое соответствующего файла. Расширения файлов приведены в данной главе в *разд. "Форматы таблиц"*.

Каждая таблица БД состоит из строк и столбцов и предназначена для хранения данных об однотипных объектах информационной системы. Строка таблицы называется *записью*, столбец таблицы — *полем* (рис. 14.1). Каждое поле должно иметь уникальное в пределах таблицы имя.

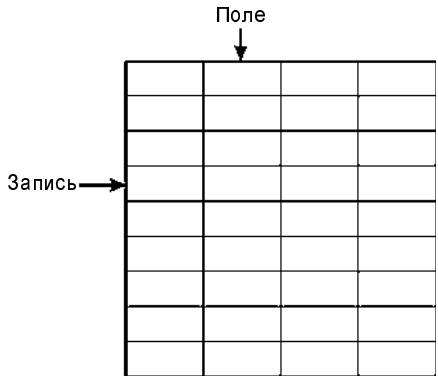


Рис. 14.1. Схема таблицы базы данных

Поле содержит данные одного из допустимых типов, например, строкового, целочисленного или типа "дата". При вводе значения в поле таблицы автоматически производится проверка соответствия типа значения и типа поля. В случае, когда эти типы не совпадают, а преобразование типа значения невозможно, генерируется исключение.

Особенности организации таблиц зависят от конкретной СУБД, используемой для создания и ведения БД. Например, в локальной таблице dBase и в таблице сервера InterBase нет поля автоинкрементного типа (с автоматически наращиваемым значением), а в таблице dBase нельзя определить ключ. Подобные особенности необходимо учитывать при выборе типа (формата) таблицы, т. к. они влияют не только на организацию БД, но и на построение приложения для работы с этой БД. Несмотря на все различия таблиц, существуют общие правила создания и ведения БД, а также разработки приложений.

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

- описание полей;
- ключ;
- индексы;
- ограничения на значения полей;
- ограничения ссылочной целостности между таблицами;
- пароли.

Иногда ограничения на значения полей, ограничения ссылочной целостности между таблицами, а также права доступа называют одним общим термином "ограничения".

Отметим, что отдельные элементы структуры зависят от формата таблиц, например, для таблиц dBase нельзя задать ограничения ссылочной целостности (т. к. у них нет ключей). Все элементы структуры задаются на физическом уровне (уровне таблицы) и действуют для всех программ, выполняющих операции с БД, включая средства разработки и ведения БД (например, программу Database Desktop). Многие из этих элементов (например, ограничения на значения полей или поля просмотра) можно также реализовать программно, однако в этом случае они действуют только в пределах своего приложения.

С таблицей в целом можно выполнять следующие операции:

- создание (определение структуры);
- изменение структуры (реструктуризация);
- переименование;
- удаление.

При *создании* таблицы задаются структура и имя таблицы. При сохранении на диске создаются все необходимые файлы, относящиеся к таблице. Их имена совпадают с именем таблицы.

При *изменении структуры* таблицы в ней могут измениться имена и характеристики полей, состав и наименования ключа и индексов, ограничения. Однако имена таблицы и ее файлов остаются прежними.

При *переименовании* таблица получает новое имя, в результате чего новое имя также получают все ее файлы. Для этого используются соответствующие программы (утилиты), предназначенные для работы с таблицами БД, например, Database Desktop или Data Pump.

Замечание

Таблицу нельзя переименовать, просто изменив названия всех ее файлов, например, с помощью Проводника Windows.

При *удалении* таблицы с диска удаляются все ее файлы. В отличие от переименования, удаление таблицы можно выполнить посредством любой программы (в том числе и с помощью Проводника Windows).

Ключи и индексы

Ключ представляет собой комбинацию полей, данные в которых однозначно определяют каждую запись в таблице. Простой ключ состоит из одного поля, а составной (сложный) — из нескольких полей. Поля, по которым построен ключ, называют *ключевыми*. В таблице может быть определен только один ключ.

Ключ обеспечивает:

- однозначную идентификацию записей таблицы;
- ускорение выполнения запросов к БД;
- установление связи между отдельными таблицами БД;
- использование ограничений ссылочной целостности.

Ключ также называют *первичным ключом* или *первичным (главным) индексом*.

Информация о ключе может храниться в отдельном файле или совместно с данными таблицы. Например, в БД Paradox для этой цели используется отдельный файл (ключевой файл или файл главного индекса) с расширением px. В БД Access вся информация содержится в одном общем файле с расширением mdb. Значения ключа располагаются в определенном порядке. Для каждого значения ключа имеется уникальная ссылка, указывающая на расположение соответствующей записи в таблице (в главном ее файле). Поэтому при поиске записи выполняется не последовательный просмотр всей таблицы, а прямой доступ к записи на основании упорядоченных значений ключа.

Ценой, которую разработчик и пользователь платят за использование такой технологии, является увеличение размера БД вследствие необходимости хранения значений ключа, например, в отдельном файле. Размер этого файла зависит не только от числа записей таблицы, но и от полей, составляющих ключ. В ключевом файле, кроме ссылок на соответствующие записи таблицы, сохраняются значения самих ключевых полей. Поэтому при вхождении в состав ключа длинных строковых полей размер ключевого файла может оказаться соизмеримым с размером файла с данными таблицы.

Таблицы различных форматов имеют свои особенности построения ключей. Вместе с тем существуют и общие правила.

- Ключ должен быть уникальным. У составного ключа значения отдельных полей (но не всех одновременно) могут повторяться.
- Ключ должен быть достаточным и не избыточным, т. е. не содержать поля, которые можно удалить без нарушения уникальности ключа.
- В состав ключа не могут входить поля некоторых типов, например, графическое поле или поле комментария.

Выбор ключевых полей не всегда является простой и очевидной задачей, особенно для таблиц с большим количеством полей. Нежелательно выбирать в качестве ключевых поля, содержащие фамилии людей в таблице сотрудников организации или названия товаров в таблице данных склада. В этом случае высока вероятность существования двух и более однофамильцев, а также товаров с одинаковыми названиями, которые различаются, к примеру, цветом (значение другого поля). Для указанных таблиц можно использовать, например, поле кода сотрудника и поле артикула товара. При этом предполагается, что указанные значения являются уникальными.

Удобным вариантом создания ключа будет использование для него поля соответствующего типа, которое автоматически обеспечивает поддержку уникальности значений. Для таблиц Paradox таким является поле автоинкрементного типа, еще одним достоинством которого является небольшой размер (4 байта). В то же время в таблицах dBase и InterBase поле подобного типа отсутствует, и программист должен обеспечивать уникальность значений ключа самостоятельно, например, используя специальные генераторы.

При создании и ведении БД правильным подходом считается задание в каждой таблице ключа даже в случае, если на первый взгляд он не нужен. В примерах таблиц, которые приводятся при изложении материала, как правило, ключ создается, и для него вводится специальное автоинкрементное поле с именем `Code` или `Number`.

Индекс, как и ключ, строится по полям таблицы, однако он может допускать повторение значений составляющих его полей — в этом и состоит его основное отличие от ключа. Поля, по которым построен индекс, называют *индексными*. Простой индекс состоит из одного поля, а составной (сложный) — из нескольких полей.

Индексы при их создании именуются. Как и в случае с ключом, в зависимости от СУБД индексы могут храниться в отдельных файлах или совместно с данными. Создание индекса называют *индексированием таблицы*.

Использование индекса обеспечивает:

- увеличение скорости доступа к данным (поиска);
- сортировку записей;
- установление связи между отдельными таблицами БД;
- использование ограничений ссылочной целостности.

В двух последних случаях индекс применяется совместно с ключом второй таблицы. Как и ключ, индекс представляет собой своеобразное оглавление таблицы, просмотр которого выполняется перед обращением к ее записям. Таким образом, использование индекса повышает *скорость доступа* к данным в таблице за счет того, что доступ выполняется не последовательным, а индексно-последовательным методом.

Сортировка представляет собой упорядочивание записей по полю или группе полей в порядке возрастания или убывания их значений. Можно сказать, что индекс служит для сортировки таблиц по индексным полям. В частности, в приложениях работы с БД по технологии BDE.NET 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.

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 нежелательно.

Таким образом, использование ключей и индексов позволяет:

- однозначно идентифицировать записи;
- избегать дублирования значений в ключевых полях;
- выполнять сортировку таблиц;
- ускорять операции поиска в таблицах;
- устанавливать связи между отдельными таблицами БД;
- использовать ограничения ссылочной целостности.

Одной из основных задач БД является обеспечение *быстрого доступа к данным* (поиска данных). Время доступа к данным в значительной степени зависит от используемых для поиска данных методов и способов.

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, его также называют *SQL-ориентированным*. Этот способ доступа ориентирован на выполнение операций с удаленными БД и является предпочтительным при работе с ними, хотя его можно использовать и для локальных БД.

Способ доступа к данным выбирается программистом и зависит от средств доступа к БД, используемых при разработке приложения. Например, в приложениях VCL.NET реализацию навигационного способа доступа можно осуществить посредством компонентов типа `TTable` или `TQuery`, а реляционного — с помощью компонента типа `TQuery`.

Связь между таблицами

В частном случае БД может состоять из одной таблицы, содержащей, например, дни рождения сотрудников организации. Однако обычно реляционная БД состоит из набора взаимосвязанных таблиц. Организация связи (отношений) между таблицами называется *связыванием* или *соединением таблиц*.

Связи между таблицами можно устанавливать как при создании БД, так и при выполнении приложения, используя средства, предоставляемые СУБД. Связывать можно две или несколько таблиц. В реляционной БД, помимо связанных, могут быть и отдельные таблицы, не соединенные ни с одной другой таблицей. Это не меняет сути реляционной БД, которая содержит единую информацию об информационной системе, связанную не в буквальном смысле (связь между таблицами), а в функциональном смысле — вся информация относится к одной системе.

Для связывания таблиц используются *поля связи* (иногда применяется термин "совпадающие поля"). Поля связи обязательно должны быть индексированными. В подчиненной таблице для связи с главной таблицей задается индекс, который также называется *внешним ключом*. Состав полей этого индекса должен полностью или частично совпадать с составом полей индекса главной таблицы.

Особенности использования индексов зависят от формата связываемых таблиц. Так, для таблиц dBase индексы строятся по одному полю и нет деления на ключ (главный или первичный индекс) и индексы. Для организации связи в главной и подчиненной таблицах выбираются индексы, составленные по полям совпадающего типа, например, целочисленного.

Для таблиц Paradox в качестве полей связи главной таблицы должны использоваться поля ключа, а для подчиненной таблицы — поля индекса. Кроме того, в подчиненной таблице обязательно должен быть определен ключ. На рис. 14.2 показана схема связи между таблицами БД Paradox.

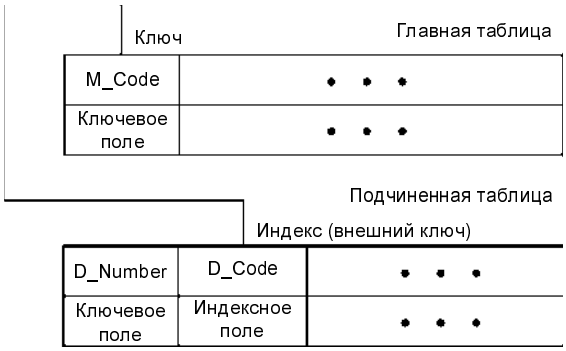


Рис. 14.2. Схема связи между таблицами базы данных Paradox

В главной таблице определен ключ, построенный по полю `M_Code` автоинкрементного типа. В подчиненной таблице определен ключ по полю `D_Number` также автоинкрементного типа и индекс, построенный по полю `D_Code` целочисленного типа. Связь между таблицами устанавливается по полям `D_Code` и `M_Code`. Индекс по полю `D_Code` является внешним ключом. В названия полей включены префиксы, указывающие на принадлежность поля соответствующей таблице. Так, названия полей главной таблицы начинаются с буквы `M` (Master), а названия полей подчиненной таблицы начинаются с буквы `D` (Detail). Подобное именование полей облегчает ориентацию в их названиях, особенно при большом количестве таблиц.

Замечание

Как отмечалось, поля связи должны быть индексированными, хотя 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 "многие-к-одному";
- отношение "многие-ко-многим".

Наиболее часто используется отношение *"один-ко-многим"*, которое означает, что одной записи главной таблицы в подчиненной таблице может соответствовать несколько записей, в том числе ни одной. После установления связи между таблицами при перемещении на какую-либо запись в главной таблице в подчиненной таблице автоматически становятся доступными записи, у которых значение поля связи равно значению поля связи текущей записи главной таблицы. Такой отбор записей подчиненной таблицы является своего рода фильтрацией.

Типичным примером является 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) поля связи может нарушиться связь между записями двух таблиц. Поэтому при редактировании поля связи записи главной таблицы нужно соответственно изменять и значения поля связи всех подчиненных таблиц;
- при удалении записи главной таблицы нужно удалять и соответствующие ей записи в подчиненной таблице (каскадное удаление);
- при добавлении записи в подчиненную таблицу значение ее поля связи должно быть установлено равным значению поля связи главной таблицы.

Ограничения по установке, изменению полей связи и каскадному удалению записей могут быть наложены на таблицы при их создании. Эти ограничения, наряду с другими элементами, например, описаниями полей и индексов, входят в структуру таблицы и действуют для всех приложений, которые выполняют операции с БД. Указанные ограничения можно задать при создании или реструктуризации таблицы, например, в среде программы Database Desktop, которая позволяет устанавливать связи между таблицами при их создании.

Ограничения, связанные с установкой, изменением значений полей связи и каскадным удалением записей, могут и не входить в структуру таблицы (таблиц), а реализовываться программным способом. В этом случае программист должен обеспечить:

- организацию связи между таблицами;
- установку значения поля связи подчиненной таблицы (это может также выполняться автоматически);
- контроль (запрет) редактирования полей связи;
- организацию (запрет) каскадного удаления записей.

Например, в случае удаления записи из главной таблицы программист должен проверить наличие соответствующих записей в подчиненной таблице. Если такие записи есть, то необходимо удалить и их или, наоборот, запретить удаление записей из обеих таблиц. И в том и в другом случае пользователю должно быть выдано предупреждение.

Механизм транзакций

Информация БД в любой момент времени должна быть целостной и непротиворечивой. Одним из путей обеспечения этого является использование механизма транзакций.

Транзакция представляет собой выполнение последовательности операций. При этом возможны две ситуации.

- Успешно завершены все операции. В этом случае транзакция считается успешной, и все изменения в БД, которые были произведены в рамках транзакции отдельными операциями, подтверждаются. В результате БД переходит из одного целостного состояния в другое.
- Неудачно завершена хотя бы одна операция. При этом вся транзакция считается неуспешной, и результаты выполнения всех операций (даже успешно выполненных) отменяются. В результате происходит возврат БД в состояние, в котором она находилась до начала транзакции.

Таким образом, успешная транзакция переводит БД из одного целостного состояния в другое. Использование механизма транзакций необходимо:

- при выполнении последовательности взаимосвязанных операций с БД;
- при многопользовательском доступе к БД.

Транзакция может быть неявной или явной. *Неявная* транзакция стартует автоматически, а по завершении также автоматически подтверждается или отменяется. *Явной* транзакцией управляет программист с использованием компонента `Database` и/или средств SQL.

Часто в транзакцию объединяются операции над несколькими таблицами, когда производятся действия по внесению в разные таблицы взаимосвязанных изменений. Пусть осуществляется перенос записей из одной таблицы в другую. Если запись сначала удаляется из первой таблицы, а затем заносится во вторую таблицу, то при сбое, например из-за перерыва в энергопитании компьютера, возможна ситуация, когда запись уже удалена, но во вторую таблицу не попала. Если запись сначала заносится во вторую таблицу, а потом удаляется из первой таблицы, то при сбое возможна ситуация, когда запись будет находиться в двух таблицах. В обоих случаях имеет место нарушение целостности и непротиворечивости БД.

Для предотвращения подобной ситуации операции удаления записи из одной таблицы и занесения ее в другую таблицу объединяются в одну транзакцию. Выполнение этой транзакции гарантирует, что при любом ее результате целостность БД нарушена не будет.

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.

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.

Форматы таблиц

Delphi .NET 2006 не имеет своего формата таблиц, но поддерживает как собственные два типа *локальных* таблиц — dBase и Paradox. Каждая из этих таблиц имеет свои особенности.

Таблицы dBase являются одним из первых появившихся форматов таблиц для персональных компьютеров и поддерживаются многими системами, которые связаны с разработкой и обслуживанием приложений, работающих с БД. Основные достоинства таблиц dBase: простота использования 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. 14.1 содержится список типов полей для таблиц Paradox 7. Для каждого типа приводятся символ, используемый для обозначения этого типа в программе Database Desktop, и описание значений, которые могут содержать поле рассматриваемого типа.

Таблица 14.1. *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

Таблица 14.1 *(окончание)*

Тип	Обозначение	Описание значения
Memo	M	Строка символов. Длина не ограничена. Первые 240 символов хранятся в файле таблицы, остальные в файле с расширением mb
Formatted Memo	F	Строка символов. Отличается от типа Memo тем, что строка может содержать форматированный текст
Graphic	G	Графическое изображение. Форматы BMP, PCX, TIFF, GIF и EPS. При загрузке в поле изображение преобразуется к формату BMP. Для хранения изображения используется файл с расширением mb
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

Имя поля в таблице Paradox должно состоять из букв (допускается кириллица) и цифр и начинаться с буквы. Максимальная длина имени составляет 25 символов. В имени можно использовать такие символы, как пробел, #, $ и некоторые другие. Не рекомендуется использовать символы ., ! и |, т. к. они зарезервированы в Delphi для других целей.

При задании ключевых полей они должны быть первыми в структуре таблицы.

Замечание

Если требуется обеспечить перенос или совместимость данных из таблиц Paradox с таблицами других форматов, желательно выбирать имя поля длиной не более 10 символов и составлять его из латинских букв и цифр.

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 звездочкой (*) файлы создаются только если в них есть необходимость; конкретная таблица может не иметь всех приведенных файлов.

Инструменты

Система Delphi .NET 2006 обеспечивает поддержку большого количества различных СУБД — как локальных (например, dBase), так и промышленных (например, Sybase или InterBase). Средства Delphi, предназначенные для работы с БД, можно разделить на два вида:

- инструменты;
- компоненты.

К *инструментам* относятся специальные программы и пакеты, обеспечивающие обслуживание БД вне разрабатываемых приложений.

Компоненты предназначены для создания приложений, осуществляющих операции с БД.

Напомним, что в Delphi .NET 2006 имеется окно Проводника данных (**Data Explorer**).

Для операций с БД система Delphi .NET 2006 предлагает приведенный ниже набор инструментов.

- Data Explorer — Проводник данных, который доступен как внешняя программа. Как и при доступе из среды системы, позволяет для различных серверов баз данных (Interbase, Oracle, DB2 и MSSQL) выполнять просмотр различных объектов баз данных (таблиц, хранимых процедур, триггеров, индексов). С помощью контекстного меню Проводника данных можно выполнять операции с соединениями баз данных (обновление, добавление, переименование и удаление).

- Borland Database Engine (BDE) — процессор баз данных, который представляет собой набор динамических библиотек и драйверов, предназначенных для организации доступа к БД из Delphi-приложений. BDE является центральным звеном при организации доступа к данным.
- BDE Administrator — утилита для настройки различных параметров BDE, настройки драйверов баз данных, создания и удаления драйверов ODBC, создания и обслуживания псевдонимов.
- dbExpress — набор драйверов для доступа к базам данных SQL с помощью таких компонентов, как `SQLConnection`, `SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`. dbExpress включает в свой состав следующие драйверы:
  - Informix (файл dbexpinf.dll);
  - InterBase (файл dbexpint.dll);
  - DB2 (файл dbexpdb2.dll);
  - Oracle (файл dbexpora.dll);
  - MS SQL (файл dbexpmss.dll);
  - MySQL (файл dbexpmys.dll).

Одни инструменты, например, BDE Administrator, можно использовать для работы с локальными и удаленными БД, другие, например, драйверы dbExpress — для работы с удаленными БД.

Состав инструментальных средств в Delphi .NET 2006/2005 стал заметно меньше, чем в последней версии продукта Delphi 7, которая не ориентирована на технологию .NET. При создании приложений для работы с БД в среде Delphi .NET 2006, на наш взгляд, удобно использовать некоторые из них. Например, при создании таблиц баз данных удобно использовать программу Database Desktop.

Создание информационной системы

Работа по созданию информационной системы включает два основных этапа:

- создание БД;
- создание приложения.

В простейшем случае БД состоит из одной таблицы. Если таблицы уже имеются, то первый этап не выполняется.

Создание таблиц базы данных

Создание таблицы и изменение структуры таблицы БД можно выполнить программно с помощью операторов языка SQL или с помощью некоторого инструмента, например программы Database Desktop.

Для создания таблицы служит оператор SQL следующего формата:

```
CREATE TABLE <имя таблицы>
                (<имя столбца> <тип данных> [NOT NULL]
                  [,<имя  столбца> <тип данных> [NOT NULL]] ... )
```

Обязательными операндами являются имя создаваемой таблицы и имя хотя бы одного столбца (поля) с указанием типа данных, хранимых в этом столбце.

При создании таблицы для отдельных полей могут указываться некоторые дополнительные правила контроля вводимых в них значений. Конструкция `NOT NULL` (не пустое) служит именно таким целям и для столбца таблицы означает, что в этом столбце должно быть определено значение.

Например, пусть требуется создать таблицу `goods` описания товаров, имеющую поля: `type` — вид товара, `comp_id` — идентификатор компании-производителя, `name` — название товара и `price` — цена товара. Оператор определения таблицы может иметь следующий вид:

```
CREATE TABLE goods (type SQL_CHAR(8) NOT NULL, comp_id SQL_CHAR(10) NOT NULL,
name SQL_VARCHAR(20), price SQL_DECIMAL(8,2)).
```

Для изменения структуры таблицы может использоваться оператор SQL следующего формата:

```
ALTER TABLE <имя таблицы>
     ( {ADD, MODIFY, DROP} <имя столбца> [<тип данных>]
        [NOT NULL]
        [,{ADD, MODIFY, DROP} <имя столбца> [<тип данных>]
        [NOT NULL]] ... )
```

Изменение структуры таблицы может состоять в добавлении (`ADD`), изменении (`MODIFY`) или удалении (`DROP`) одного или нескольких столбцов таблицы. Правила записи оператора `ALTER TABLE` такие же, как и оператора `CREATE TABLE`. При удалении столбца указывать `<тип данных>` не нужно.

Например, пусть в созданной ранее таблице `goods` необходимо добавить поле `number`, отводимое для хранения величины запаса товара. Для этого следует записать оператор вида:

```
ALTER TABLE goods (ADD number SQL_INTEGER)
```

Для работы с таблицами БД при разработке приложения удобно использовать программу Database Desktop, которая поставляется вместе с Delphi 7. Эта программа позволяет:

- ❒ создавать таблицы;
- ❒ изменять структуры;
- ❒ редактировать записи.

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

Замечание

Почти все рассматриваемые далее действия по управлению структурой таблицы можно выполнить также программно в процессе выполнения приложения.

Процесс создания новой таблицы начинается с вызова команды **File | New | Table** (Файл | Новая | Таблица) и происходит в интерактивном режиме. При этом разработчик должен:

- выбрать формат (тип) таблицы;
- задать структуру таблицы.

В начале создания новой таблицы в окне **Create Table** (Создание таблицы) 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.

После выбора формата таблицы появляется окно определения структуры таблицы (рис. 14.3), в котором выполняются следующие действия:

- описание полей;
- задание ключа;
- задание индексов;

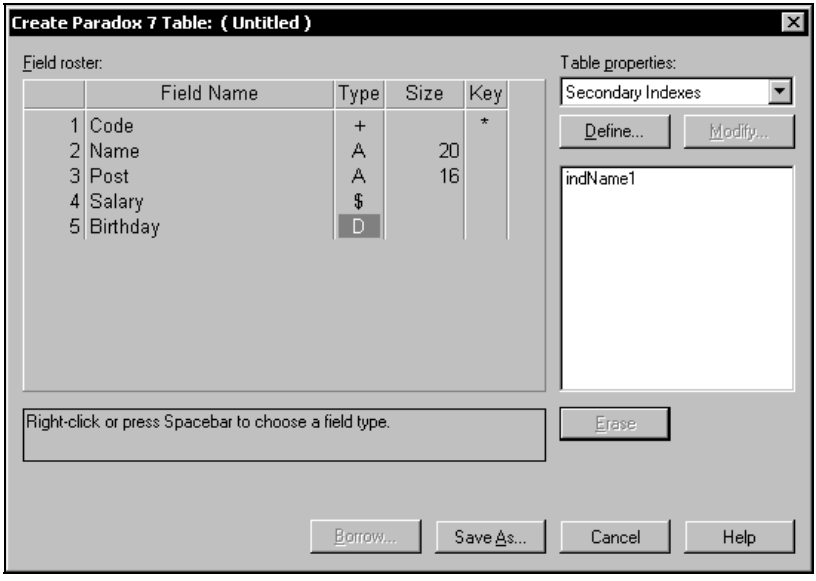


Рис. 14.3. Определение структуры таблицы

- 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, таких как задание ключа и паролей, производится только для таблиц определенных форматов, например для таблиц Paradox.

После определения структуры таблицы ее необходимо сохранить, нажав кнопку **Save As** и указав расположение таблицы на диске и ее имя. В результате на диск записывается новая таблица, первоначально пустая, при этом все необходимые файлы создаются автоматически.

Описание полей

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

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

Имя поля вводится по правилам, установленным для выбранного формата таблиц. Правила именования и допустимые типы полей таблиц Paradox описаны в начале этой главы.

Тип поля можно задать, непосредственно указав соответствующий символ, например `A` для символьного или `I` для целочисленного поля, или выбрать его в списке, раскрываемом нажатием клавиши <Space> или щелчком правой кнопки мыши в столбце **Type**. Список содержит все типы полей, допустимые для заданного формата таблицы. В списке подчеркнуты символы, используемые для обозначения соответствующего типа, при выборе типа эти символы автоматически заносятся в столбец **Type**.

Размер поля задается не всегда, необходимость его указания зависит от типа поля. Для полей определенного типа, например автоинкрементного (`+`) или целочисленного (`I`), размер поля не задается. Для поля строкового типа размер определяет максимальное число символов, которые могут храниться в поле.

Добавление к списку полей новой строки выполняется переводом курсора вниз на несуществующую строку, в результате чего эта строка появляется в конце списка. Вставка новой строки между существующими строками с уже описанными полями выполняется нажатием клавиши <Insert>. Новая строка вставляется перед строкой,

в которой расположен курсор. Для удаления строки необходимо установить курсор на эту строку и нажать комбинацию клавиш <Ctrl>+<Delete>.

Ключ создается указанием его полей. Для указания ключевых полей в столбце ключа (**Key**) нужно установить символ *, переведя в эту позицию курсор и нажав любую алфавитно-цифровую клавишу. При повторном нажатии клавиши отметка принадлежности поля ключу снимается. В структуре таблицы ключевые поля должны быть первыми, т. е. верхними в списке полей. Часто для ключа используют автоинкрементное поле (см. рис. 14.3).

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** (вторичные индексы);
- **Validity Checks** (проверка правильности ввода значений полей) — выбирается по умолчанию;
- **Referential Integrity** (ссылочная целостность);
- **Password Security** (пароли);
- **Table Language** (язык таблицы, языковой драйвер);
- **Table Lookup** (таблица выбора);
- **Dependent Tables** (подчиненные таблицы).

После выбора какого-либо пункта этого списка в правой части окна определения структуры таблицы появляются соответствующие элементы, с помощью которых выполняются дальнейшие действия.

Состав данного списка зависит от формата таблицы. Так, для таблицы dBase он содержит только пункты **Indexes** и **Table Language**.

Задание индексов

Задание индекса сводится к определению:

- состава полей;
- параметров;
- имени.

Эти элементы устанавливаются или изменяются при выполнении операций создания, изменения и удаления индекса. Напомним, что для таблиц Paradox индекс называют также вторичным индексом.

Для выполнения операций, связанных с заданием индексов, необходимо выбрать пункт **Secondary Indexes** (Вторичные индексы) списка **Table properties** (Свойст-

ва таблицы), при этом под списком появляются кнопки **Define** (Определить) и **Modify** (Изменить), список индексов и кнопка **Erase** (Удалить). В списке индексов выводятся имена созданных индексов, на рис. 14.3 это индекс `indName1`.

Создание нового индекса начинается с нажатия кнопки **Define**, которая всегда доступна. Она открывает окно **Define Secondary Index** (Задание вторичного индекса), в котором задаются состав полей и параметры индекса (рис. 14.4).

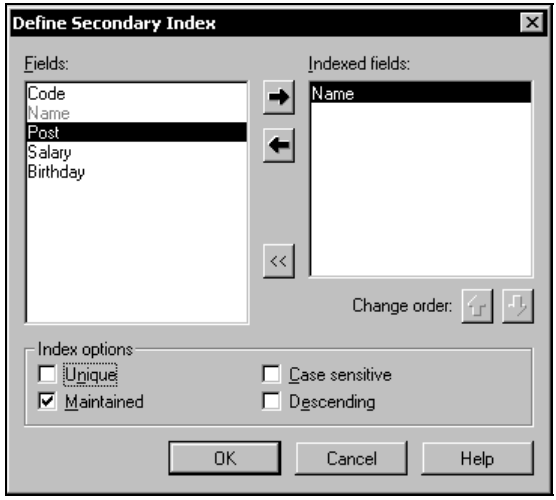


Рис. 14.4. Окно задания индекса

В списке **Fields** окна выводятся имена всех полей таблицы, включая и те, которые недопустимы в составе индекса, например, графическое поле или поле комментария. В списке **Indexed fields** (Индексные поля) содержатся поля, которые включаются в состав создаваемого индекса. Перемещение полей между списками выполняется выделением нужного поля (полей) и нажатием расположенных между этими списками кнопок с изображением горизонтальных стрелок. Имена полей, которые нельзя включать в состав индекса, выделяются в левом списке серым цветом. Поле не может быть повторно включено в состав индекса, если оно уже выбрано и находится в правом списке.

Замечание

При работе с записями индексные поля обрабатываются в порядке следования этих полей в составе индекса. Это нужно учитывать при указании порядка полей в индексе.

Изменить порядок следования полей в индексе можно с помощью кнопок с изображением вертикальных стрелок, имеющих общее название **Change order** (Изменить порядок). Для перемещения поля (полей) необходимо его (их) выделить и нажать нужную кнопку.

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

- **Unique** — индекс требует для составляющих его полей уникальных значений;
- **Maintained** — задается автоматическое обслуживание индекса;
- **Case sensitive** — для полей строкового типа учитывается регистр символов;
- **Descending** — сортировка выполняется в порядке убывания значений.

Так как у таблиц dBase нет ключей, для них использование параметра **Unique** является единственной возможностью обеспечить уникальность записей на физическом уровне (уровне организации таблицы), не прибегая к программированию.

После задания состава индексных полей и нажатия кнопки **OK** появляется окно **Save Index As**, в котором нужно указать имя индекса (рис. 14.5). Для удобства обращения к индексу в его имя можно включить имена полей, указав какой-нибудь префикс, например `ind`. Нежелательно образовывать имя индекса только из имен полей, т. к. для таблиц Paradox подобная система именования используется при автоматическом образовании имен для обозначения ссылочной целостности между таблицами. После повторного нажатия кнопки **OK** сформированный индекс добавляется к таблице, и его имя появляется в списке индексов.

Созданный индекс можно изменить, определив новый состав полей, параметров и имени индекса. Изменение индекса практически не отличается от его создания. После выделения индекса в списке и нажатия кнопки **Modify** снова открывается окно задания индекса (см. рис. 14.4). При нажатии кнопки **OK** появляется окно сохранения индекса (см. рис. 14.5), содержащее имя изменяемого индекса, которое можно исправить или оставить прежним.

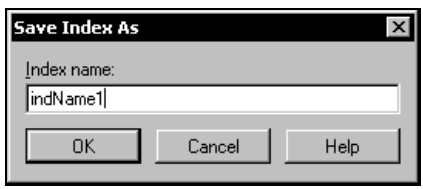


Рис. 14.5. Задание имени индекса

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

Кнопки **Modify** и **Erase** доступны, только если индекс выбран в списке.

Ограничения на значения полей

Задание ограничений на значения полей заключается в указании для полей:

- требования обязательного ввода значения;
- минимального значения;

- максимального значения;
- значения по умолчанию;
- маски ввода.

Замечание

Установленные ограничения задаются на физическом уровне (уровне таблицы) и действуют для любых программ, выполняющих операции с таблицей: как для программ типа Database Desktop, так и для приложений, создаваемых в Delphi. Дополнительно к этим ограничениям или вместо них в приложении можно задать *программные* ограничения.

Для выполнения операций, связанных с заданием ограничений на значения полей, нужно выбрать пункт **Validity Checks** (Проверка значений) комбинированного списка **Table properties** (см. рис. 14.3).

Ссылочная целостность

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

- запрещается изменять поле связи или удалять запись главной таблицы, если для нее имеются записи в подчиненной таблице;
- при удалении записи в главной таблице автоматически удаляются соответствующие ей записи в подчиненной таблице (каскадное удаление).

Для выполнения операций, связанных с заданием ссылочной целостности, необходимо выбрать пункт **Referential Integrity** комбинированного списка **Table properties** (см. рис. 14.3).

Пароль и языковой драйвер

Пароль позволяет задать права доступа пользователей (приложений) к таблице. Если для таблицы установлен пароль, то он будет автоматически запрашиваться при каждой попытке открытия таблицы.

Замечание

Пароль действует на физическом уровне и его действие распространяется на все программы, выполняющие доступ к таблице: как на программы типа Database Desktop, так и на создаваемые приложения Delphi.

Для выполнения операций, связанных с заданием пароля, нужно выбрать строку **Password Security** в комбинированном списке **Table properties** окна определения структуры таблицы (см. рис. 14.3).

Для задания языкового драйвера нужно выбрать пункт **Table Language** (Язык таблицы) комбинированного списка **Table properties** окна определения структуры таблицы (см. рис. 14.3).

Изменение структуры таблицы

Структуру существующей таблицы можно изменить, выполнив команду **Table | Restructure** после предварительного выбора таблицы в окне программы Database Desktop. В результате открывается окно определения структуры таблицы, и дальнейшие действия не отличаются от действий, выполняемых при создании таблицы.

Замечание

При изменении структуры таблицы с ней не должны работать другие приложения, в том числе Delphi .NET 2006. Поэтому предварительно необходимо закрыть систему программирования или приложение, в котором компоненты связаны с перестраиваемой таблицей. Другим вариантом является отключение активности компонентов, связанных с перестраиваемой таблицей, для чего свойству `Active` этих компонентов через Инспектор объектов устанавливается значение `False`.

Переименование таблицы следует выполнять из среды программы Database Desktop, а не из среды Windows, например, с помощью Проводника. Для этого при работе со структурой таблицы можно нажать кнопку **Save As** и задать новое имя таблицы. В результате в указанном каталоге диска появятся все необходимые файлы таблицы. При этом старая таблица также сохраняется. Информация о названии таблицы используется внутри ее файлов, поэтому простое переименование всех файлов таблицы приведет к ошибке при попытке доступа к ней.

Если необходимо просто ознакомиться со структурой таблицы, то выполняется команда **Table | Into Structure1**. В результате появляется окно определения структуры таблицы, но элементы, с помощью которых в структуру таблицы могут быть внесены изменения, заблокированы. Просмотр структуры возможен также для таблицы, с которой связаны другие приложения.

ГЛАВА 15

Технология ADO.NET

Общая характеристика

ADO.NET (ActiveX Data Object.NET) содержит набор классов, используемый для доступа к источникам данных на платформе .NET. ADO.NET представляет собой новую объектную модель, которая использует стандарт XML для передачи данных. В ADO.NET реализована идея использования *отсоединенных* наборов данных, причем такой способ работы является основным. По сравнению с ADO, ADO.NET предполагает более легкое программирование, лучшую производительность и масштабирование, меньшую зависимость от источников данных.

ADO.NET можно использовать для доступа к реляционным СУБД, таким как SQL Server 2000, и ко многим дополнительным источникам данных, для работы с которыми предназначен провайдер OLE DB.

Одно из ключевых новшеств ADO.NET — замена ADO-объекта `RecordSet` комбинацией объектов `DataTable`, `DataSet`, `DataAdapter` и `DataReader`. Объект `DataTable` представляет набор (`collection`) записей отдельной таблицы и в этом отношении аналогичен `RecordSet`. Объект `DataSet` представляет набор объектов `DataTable`, а также содержит отношения и ограничения, используемые при связывании таблиц. На самом деле `DataSet` — это хранящаяся в памяти реляционная структура данных со встроенной поддержкой XML.

В ADO.NET поддержка реляционной модели осуществляется двумя способами. *Первый* способ реализует подсоединенную модель доступа к данным, в которой функционируют стандартные методы доступа к реляционной базе данных, включая поддержку параметрических запросов, хранимых процедур, SQL-операторов пакетного выполнения и транзакций. Этот способ полностью аналогичен способам, применяющимся в известных интерфейсах доступа к данным, таких как OLE DB, ODBC, JDBC.

При *втором* способе доступ к данным осуществляется в *отсоединенном* режиме с помощью объекта `DataSet`, имитирующего базу данных. Над данными, содер-

жащимися в этом объекте, осуществимы все операции, характерные для баз данных. Объект `DataSet` может содержать любое количество реляционных таблиц, отношений и ограничений и даже позволяет получать подмножество таблиц с помощью SQL-подобного языка запросов. Кроме того, в отсоединенную модель входит объект `DataAdapter`, который играет роль связующего класса между реальной базой данных и объектом `DataSet`.

Технология ADO.NET создана для использования в управляемых проектах. Предыдущая технология ADO основана на технологии COM и при использовании из управляемых приложений требует дополнительных затрат на выполнение кода. К тому же ADO имеет меньшие возможности при работе с отключенными наборами данных и XML. Например, в ADO непросто сохранить изменения, произведенные в отключенном курсоре. Рассмотрим некоторые *преимущества* ADO.NET в сравнении с ADO.

- *Масштабируемость.* При использовании объекта `DataSet` работа происходит с отсоединенными наборами данных. Это означает, что вы используете соединение с источником данных очень короткое время. Во многих системах количество подключений к базам данных является самым узким местом в плане масштабируемости. Для таких систем ADO.NET является хорошим решением, резко повышающим их масштабируемость.
- *Независимость от источника данных.* В ADO возможности объекта `RecordSet` сильно зависели от используемого источника данных. Теоретически ADO обеспечивала доступ к данным независимо от источника данных, на практике всегда необходимо было иметь хорошее представление о возможностях провайдера. В ADO.NET `DataSet` действительно не зависим от источника данных, и изменение провайдера, с помощью которого заполняется `DataSet`, не влияет на функциональность `DataSet`.
- *Способность к взаимодействию.* Так как ADO.NET использует XML как стандартный формат передачи данных, программа, которой необходимо получить данные из компонента ADO.NET, не обязана сама быть компонентом ADO.NET. В общем случае она вообще может не быть Windows-программой. Единственное требование — эта программа должна понимать XML. И это позволяет компонентам ADO.NET при использовании других компонентов и служб легко взаимодействовать с любой программой на любой платформе.

Компоненты ADO.NET существуют в пяти главных пространствах имен в *библиотеке классов .NET*. Укажем назначение этих пространств имен.

- `System.Data` — содержит фундаментальные классы с базовой функциональностью ADO.NET. В их число входят классы `DataSet` и `DataRelation`, которые позволяют манипулировать структурированными реляционными данными.
- `System.Data.Common` — эти классы применяются другими классами в пространствах имен `System.Data.SqlClient` и `System.Data.OleDb`, которые являются их наследниками и предоставляют конкретные версии, настроенные для поставщиков SQL Server и OLE DB.

- `System.Data.OleDb` — содержат классы, которые используются для соединения с поставщиком OLE DB, включая `OleDbCommand` и `OleDbConnection`.
- `System.Data.SqlClient` — содержит классы, которые используются для соединения с базой данных Microsoft SQL Server. Эти классы предоставляют такие же свойства и методы аналогично классам из `System.Data.OleDb`. Отличие состоит в том, что они оптимизированы для SQL Server.
- `System.Data.SqlTypes` — содержит структуры для специфических типов данных SQL Server, например, `SqlMoney` и `SqlDateTime`. Эти типы не являются обязательными и используются для работы с типами данных SQL Server без необходимости конвертировать их в стандартные эквиваленты .NET. Использование таких структур позволяет увеличить производительность обмена данными за счет устранения автоматического преобразования данных.

Базовые объекты ADO.NET можно разбить на две группы: первая используется для хранения и управлениями данными (классы `DataSet`, `DataTable`, `DataRow` и `DataRelation`), вторая группа — для обеспечения соединения с источниками данных (классы `Connections`, `Commands` и `DataReader`). Объекты ADO.NET второй группы существуют в двух видах: для связи с поставщиками OLE DB и для взаимодействия с SQL Server.

Объекты данных позволяют хранить локальные данные, не имеющие связи с источником. Кроме того, они не хранят соединение с источником данных и, следовательно, их можно создавать самостоятельно, не используя базу данных. Объект `DataSet` не имеет непосредственного соединения с источником данных и его можно создавать без обращения к базе данных.

Схема доступа к данным

Одна из схем доступа к данным с помощью ADO.NET представлена на рис. 15.1. Необходимая пользователю информация находится в базе данных. Обрабатываемые данные размещаются в объекте `DataSet`, не имеющем соединения с источником данных. Тем самым достигается возможность работы с данными в течение длительного промежутка времени. На рис. 15.1 серым цветом под пунктирной линией для наглядности выделены неподсоединенные объекты.

Отметим, что технология ADO.NET представляет собой концепцию универсального доступа к данным, которую можно определить как распределенное хранение данных и управление ими.

ADO.NET в Delphi 2006

Для разработки приложений БД по технологии ADO.NET в Delphi .NET 2006 используются компоненты, расположенные на странице **Data Components** Палитры компонентов. Они принадлежат пространству имен `System.Data`. Как и для всей

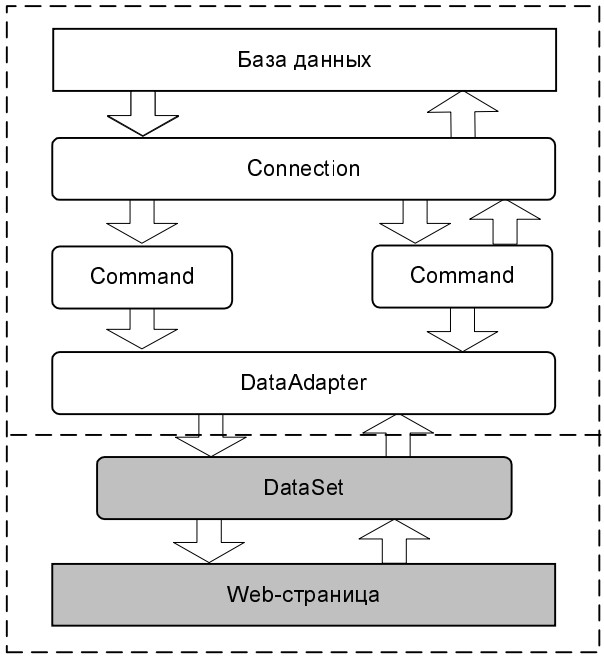


Рис. 15.1. Схема доступа к данным с помощью ADO.NET

технологии ADO.NET, наиболее общее использование компонентов **Data Components** связано с построением табличных запросов, выборкой и отображением результатов в приложении и, при необходимости, изменением данных пользователям и сохранение в базе данных результатов внесенных изменений.

На странице **Data Components** Палитры расположены следующие компоненты ADO.NET:

- `SqlDataAdapter` — представляет множество команд данных и соединения с базой данных, которые используются для заполнения компонента `DataSet` и обновления (сохранения измененных данных) базы данных SQL-сервера;
- `SqlConnection` — представляет открытое соединение с базой данных SQL-сервера;
- `DataView` — представляет связываемые данные, настроенный просмотр (view) табличного набора данных `DataTable` для сортировки, фильтрации, поиска, редактирования и навигации;
- `DataSet` — представляет внутренний кэш набора данных, полученного из источника данных;
- `SqlCommand` — представляет оператор на языке Transact-SQL или хранимую процедуру, предназначенные для выполнения на SQL Server.

Как и в случае .NET Framework, в Delphi .NET 2006 различают наборы данных `DataSet` двух типов: типизированные (typed) и нетипизированные (standard). Разница между ними заключается в том, имеется ли схема данных для `DataSet`.

Типизированный `DataSet` имеет место в случае, если при работе с набором данных сначала создается схема, а затем в него помещаются данные. При использовании типизированного `DataSet` требуется выполнить ряд вспомогательных действий по определению правил и форматов, которым должен соответствовать этот набор данных.

Создание нетипизированного объекта `DataSet` не требует дополнительных действий, поскольку он не содержит внутренней схемы размещения данных. В нетипизированном объекте `DataSet` таблицы, столбцы и другие элементы представлены как коллекции. Создание нетипизированного объекта `DataSet` проще, но при его использовании недоступны все возможности его типизированного аналога.

Достоинством применения типизированного объекта `DataSet` является возможность ссылаться на столбцы набора данных с помощью имен. При этом программирование работы с набором данных оказывается проще.

Использование нетипизированных объектов `DataSet` необходимо в случаях, когда схема набора данных отсутствует. Такое может быть, к примеру, при взаимодействии приложения с компонентом, возвращающим набор данных с неизвестной структурой. Если используемый набор данных имеет динамически перестраиваемую структуру, то использование типизированного объекта `DataSet` также может оказаться нецелесообразным.

ГЛАВА 16

Использование провайдеров BDP.NET

Провайдеры данных для .NET

Провайдеры данных для .NET Framework при использовании технологии ADO.NET служат для соединения с базой данных, выполнения команд и получения результатов. Получаемые результаты обрабатываются непосредственно или помещаются в компонент ADO.NET `DataSet` для обеспечения возможности доступа к ним со стороны пользователя совместно с данными из других источников. При разработке провайдеров .NET Framework ставилась цель обеспечить легковесный, минимальный промежуточный слой между источником данных и кодом приложения, способный обеспечить увеличение производительности без снижения функциональности.

Платформа .NET Framework версии 1.1 в своем составе включает провайдеры для MS SQL Server, OLE DB, ODBC и Oracle. Эти провайдеры обеспечивают следующее:

- MS SQL Server — поддерживает доступ к одноименному серверу фирмы Microsoft. Рекомендуется использовать для создания одно- и многозвенных приложений для MS SQL Server версии 7.0 и выше. Размещается в пространстве имен `System.Data.SqlClient`.
- OLE DB — поддерживает доступ к источникам данных с помощью одноименного интерфейса. Рекомендуется для создания многозвенных приложений для MS SQL Server версии 6.5 и ниже или для большинства провайдеров, поддерживающих интерфейс OLE DB. Размещается в пространстве имен `System.Data.OleDb`.
- ODBC — поддерживает доступ к источникам данных ODBC. Рекомендуется для создания одно- и многозвенных приложений с источниками данных ODBC. Размещается в пространстве имен `System.Data.Odbc`.

- Oracle — поддерживает доступ к данным сервера Oracle. Рекомендуется для создания одно- и многозвенных приложений с источниками данных Oracle. Размещается в пространстве имен `System.Data.OracleClient`.

В системе программирования Delphi .NET 2006 в дополнение к этим провайдерам имеются провайдеры данных фирмы Borland для Microsoft .NET (Borland Data Providers for .NET, BDP.NET). Провайдеры BDP.NET представляют собой реализацию .NET-провайдера и соединяются с наиболее популярными базами данных.

Провайдеры данных BDP.NET

Достоинством использования провайдеров BDP.NET является то, что они обеспечивают высокую производительность при доступе к источникам данных без промежуточного слоя COM (рис. 16.1).

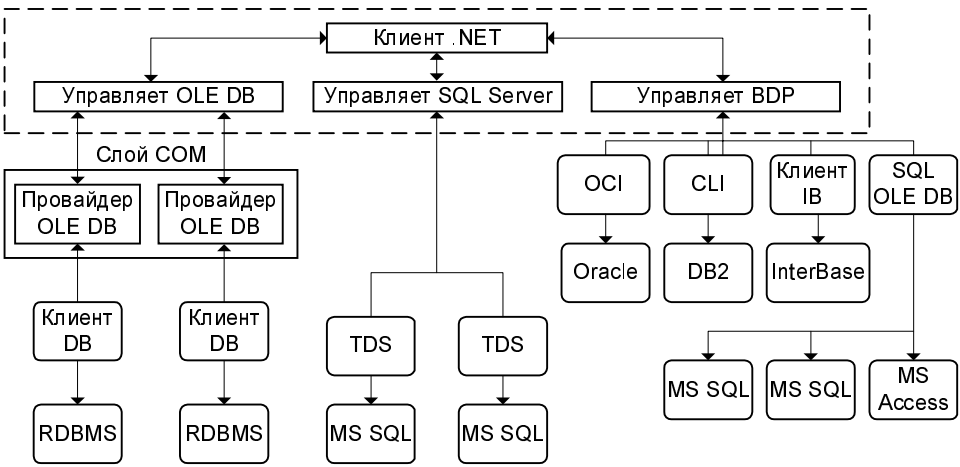


Рис. 16.1. Схема использования провайдеров данных

Для BDP.NET компоненты, доступ к метаданным и конструкторы определены под следующими пространствами имен:

- `Borland.Data.Provider` — содержит объекты, общие для всех провайдеров BDP, включая объекты ошибок, объекты классов исключений, перечислители типов данных, параметры провайдера, и интерфейсы для построения собственных классов `Command`, `Connection` и `Cursor`;
- `Borland.Data.Common` — содержит ключевые классы BDP.NET, например, `BdpCommand`, `BdpConnection`, `BdpDataAdapter` и др., которые обеспечивают средства для взаимодействия с внешними источниками данных, такими как базы данных Oracle, DB2, InterBase и MS SQL Server;

- `Borland.Data.Schema` — содержит интерфейсы для создания собственных классов манипулирования схемой базы данных, таких как ряд типов и перечислителей, которые определяют метаданные;
- `Borland.Data.Design` — содержит конструкторы.

Компоненты BDP.NET расположены на странице **Borland Data Providers** Палитры компонентов:

- `BdpConnection` — служит для создания одноименного объекта соединения с базой данных;
- `BdpCommand` — служит для создания одноименного объекта, обеспечивающего выполнение SQL-запросов и хранимых процедур;
- `BdpDataAdapter` — задает объект, представляющий собой мост между объектом `DataSet` и базой данных в его основе;
- `BdpCommandBuilder` — генерирует однотабличные команды как часть операции объекта `BdpDataAdapter`;
- `BdpCopyTable` — служит для копирования таблицы (структуры, первичного ключа и данных) из одного источника данных BDP.NET в другой;
- `DataHub` — служит проводником между наборами данных `DataSet` и `DataSync`;
- `DataSync` — поддерживает реальные данные от любого провайдера данных .NET на этапе проектирования приложения;
- `RemoteConnection` — обеспечивает соединение между клиентом и удаленным сервером;
- `RemoteServer` — удаленный сервер данных, управляющий запросами данных для многих клиентов в двухзвенных и трехзвенных приложениях.

BDP.NET включает провайдеров для ряда популярных систем управления базами данных, каждому из которых соответствует свое пространство имен:

- InterBase — `Borland.Data.InterBase`;
- Oracle — `Borland.Data.Oracle`;
- IBM DB2 — `Borland.Data.Db2`;
- Microsoft SQL Server — `Borland.Data.Mssql`;
- Microsoft Access — `Borland.Data.Msacc`;
- Sybase — `Borland.Data.Sybase`.

Провайдеры BDP.NET устанавливают соответствие типов данных SQL к типам данных .NET Framework, предотвращая тем самым необходимость изучения типов данных конкретной базы данных. В зависимости от базы данных BDP.NET приводит типы данных к их родным типам данных. Там, где это применимо, BDP.NET обеспечивает:

- установление соответствия совместимых типов по базам данных;
- логические типы данных приводятся к родным типам .NET.

Класс `DataSet` в ADO.NET использует типы данных .NET Framework. Под типы данных BDP.NET задаются типы данных .NET для поддерживаемых баз данных.

Платформа .NET Framework включает широкий диапазон логических типов данных. BDP.NET наследует логические типы данных, обеспечивая встроенное установление соответствия для поддерживаемых баз данных: DB2, InterBase, MS SQL, MSDL и Oracle.

В качестве примера схема соответствия типов данных BDP.NET для сервера InterBase приведена в табл. 16.1.

Таблица 16.1. *Схема соответствия типов для InterBase*

InterBase	BDP	BdpSub	System.
CHAR	String	stFixed	String
VARCHAR	String	NA	String
SMALLINT	Int16	NA	Int16
INTEGER	Int32	NA	Int32
FLOAT	Float	NA	Single
DOUBLE	Double	NA	Double
BLOB Sub_Type 0	Blob	stBinary	Byte[]
BLOB Sub_Type 1	Blob	stMemo	Char[]
TIMESTAMP	Datetime	NA	DateTime

Аналогичные схемы соответствия типов данных имеются и для других баз данных (DB2, MS SQL, MSDL и Oracle).

Приложение Windows Forms с ADO.NET и BDP.NET

Напомним, что создание приложения Windows Forms в среде Delphi .NET 2006 выполняется по команде **File | New | Windows Forms Application – Delphi for .NET**.

Рассмотрим технологию создания простейшего приложения ADO.NET для работы с базами данных с использованием Windows Forms и BDP.NET. После генерации требуемых объектов соединения проект отображает данные в DataGrid. Для облегчения разработки приложений для баз данных BDP.NET включает компонент Конструктор. При этом вместо того, чтобы помещать индивидуальные компоненты в Конструктор и поочередно настраивать их, целесообразно использо-

вать конструкторы BDP.NET, чтобы быстро создать и настроить компоненты базы данных.

Рассмотрим применение основных компонентов Windows Forms, ADO.NET и BDP.NET. Чтобы подтвердить и конфигурировать провайдер данных, можно перетащить (Drag and Drop) объекты из Проводника данных (Data Explorer), который представляет собой окно с деревом баз данных различных провайдеров, расположенное в правой части пользовательского интерфейса среды Delphi (рис. 16.2).

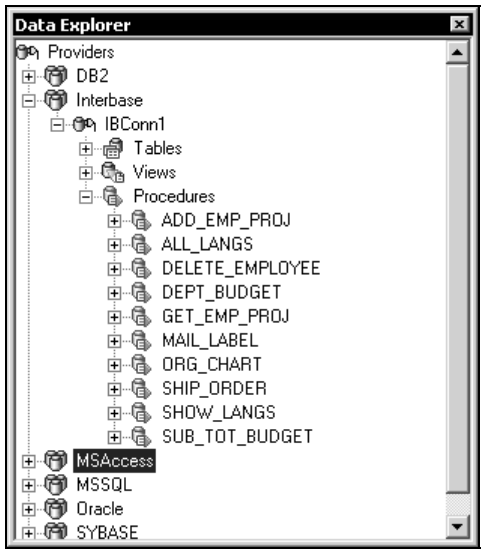


Рис. 16.2. Вид Проводника данных

Построение проекта BDP.NET включает три основных шага:

1. Настройка компонентов соединения BDP.NET и источника данных.
2. Создание и настройка компонента адаптера `BdpDataAdapter`.
3. Связывание компонента сетки данных `DataGrid` с компонентами соединения.

Чтобы настроить компоненты соединения и источник данных, нужно выполнить следующее:

1. Задав команду **File | New | Windows Forms Application – Delphi for .NET**, вызвать Конструктор Windows Forms.
2. Перетащить компонент `BdpConnection` в нижнюю часть (component tray) Конструктора со страницы **Borland Data Provider** Палитры компонентов.
3. В нижней части Инспектора объектов выполнить щелчок в строке **Connection Editor**.

4. При необходимости добавить новое соединение:
   - выполнить щелчок по кнопке **Add**;
   - в открывшемся окне выбрать тип провайдера в раскрывающемся списке **Provider Name**;
   - ввести имя соединения и нажать **OK**.
5. Выбрать соединение.
6. В строке **Database** поля **Connection Settings** задать путь к базе данных и нажать **OK**.

Замечание

При использовании сервера InterBase база данных может размещаться в локальном каталоге c:\Program Files\Common Files\Borland Shared\Data. При соединении с разделяемой сетью нужно указать сетевой путь и необходимо иметь права доступа к удаленному серверу.

7. Заполнить поля **UserName** и **Password** для базы данных. Напомним, что для базы данных сервера InterBase в них указываются значения `SYSDBA` и `masterkey`.
8. Нажать кнопку **Test** для проверки правильности соединения.
9. При получении сообщения об успешности соединения нажать **OK**.

Для создания и настройки адаптера данных (компонента `BdpDataAdapter`) нужно выполнить следующие шаги:

1. Со страницы **Borland Data Provider** Палитры компонентов поместить в нижнюю часть Конструктора компонент `BdpDataAdapter`.
2. В окне Инспектора объектов раскрыть свойство `SelectCommand` в области заполнения.
3. Выбрать объект соединения из раскрывающегося списка в строке свойства `Connection`.
4. Выполнить щелчок в строке **Configure Data Adapter** в нижней части Инспектора объектов.
5. В открывшемся диалоговом окне редактора **Data Adapter Configuration** (рис. 16.3) на вкладке **Command** выбрать имя таблицы, выделить нужные поля таблицы и нажать кнопку **Generate SQL**.

Замечания

- С помощью вкладки **Preview Data** редактора **Data Adapter Configuration** можно выполнить предварительный просмотр набора данных, задаваемого автоматически сформированным оператором `SELECT`. Для этого нужно нажать кнопку **Refresh**. В результате в окне предварительного просмотра будут отображаться записи выбранных полей таблицы.

- На вкладке **DataSet** рассматриваемого редактора путем установки переключателя **New DataSet** или **Existing DataSet** можно запросить соответственно автоматическое создание нового компонента `DataSet` (с указанием нужного имени) или связать настраиваемый адаптер (компонент `BdpDataAdapter`) с имеющимся компонентом `DataSet`.
- Для создания нового компонента `DataSet` выберем вкладку **DataSet**, установим переключатель **New DataSet**, уточним имя компонента или оставим имя по умолчанию и нажмем **OK**. В результате новый компонент `DataSet` появится среди уже имеющихся компонентов в нижней части среды Delphi.

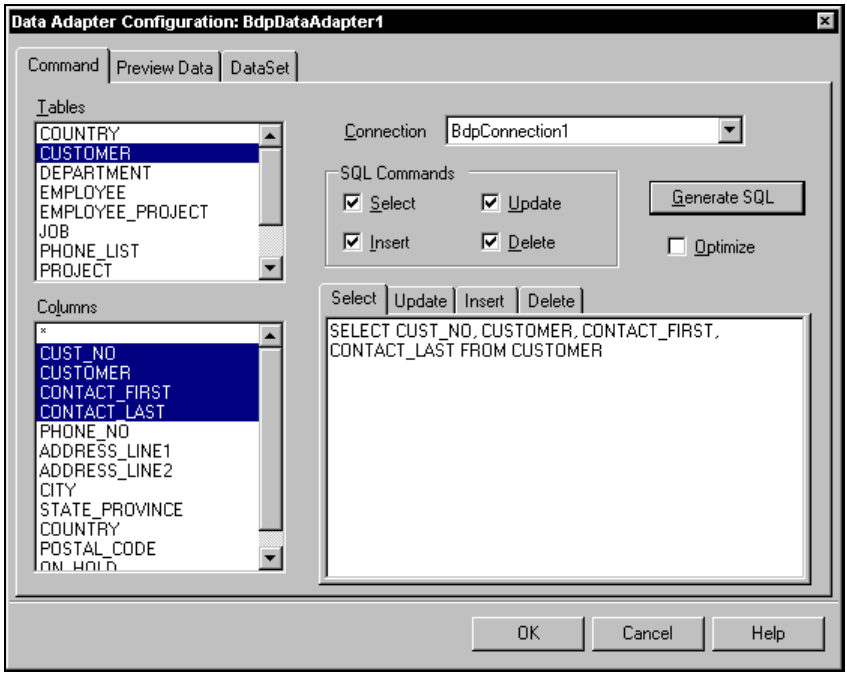


Рис. 16.3. Окно редактора **Data Adapter Configuration**

Для связывания компонента `DataGrid` с компонентами соединения нужно выполнить следующее.

1. С помощью Инспектора объектов свойству `Active` компонента `BdpDataAdapter` установить значение `True`.
2. Со страницы **Data Controls** Палитры компонентов перетащить компонент `DataGrid` в Конструктор.
3. С помощью Инспектора объектов свойству `DataSource` объекта `DataGrid` выбрать из раскрывающегося списка значение имени сгенерированного нами ранее объекта `DataSet` (по умолчанию `DataSet1`).

4. Кроме того, свойству `DataMember` объекта `DataGrid` выбрать из раскрывающегося списка имя соответствующей таблицы.

В результате компонент `DataGrid` будет отображать данные из набора `DataSet` (рис. 16.4).

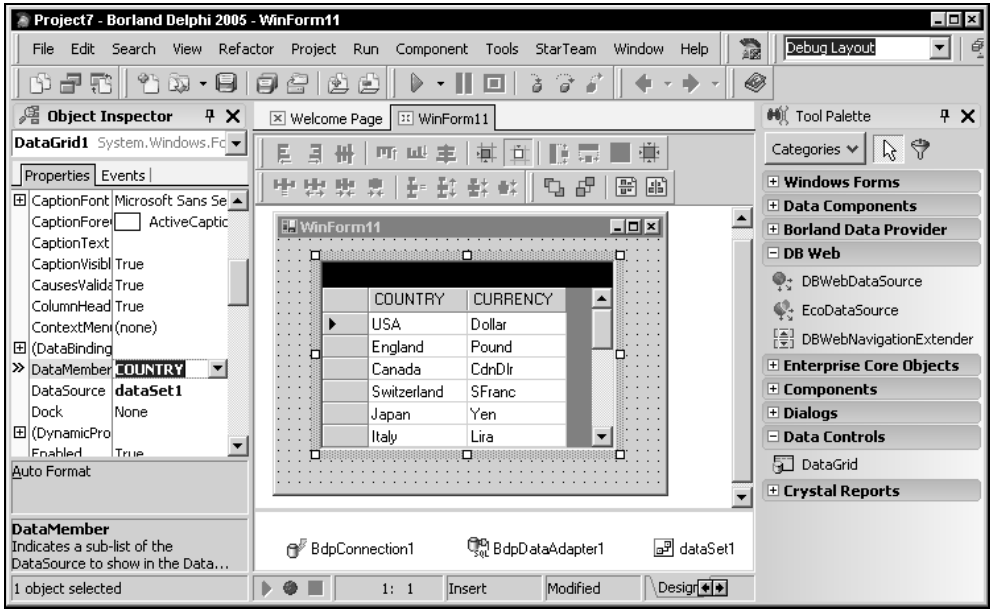


Рис. 16.4. Вид приложения ADO.NET с BDP.NET при проектировании

Теперь нам остается задать команду **Run** | **Run**, чтобы запустить приложение на компиляцию и выполнение. При выполнении приложение будет отображать с помощью компонента `DataGrid` данные из таблицы.

Приведенное нами приложение Windows Forms, хотя и является простейшим в смысле его функциональности, но позволяет получить представление о составе и взаимодействии основных компонентов при использовании технологии ADO.NET и провайдеров BDP.NET.

ГЛАВА 17

Подготовка отчетов Rave Reports

Отчет представляет собой печатный документ, содержащий данные, аналогичные получаемым в результате выполнения запроса к БД или из некоторого другого источника — электронной таблицы, сообщения электронной почты, текстового документа и др. Можно выделить следующие виды отчетов:

- простой отчет;
- отчет с группированием данных;
- отчет для таблиц, связанных отношением "главный-подчиненный";
- составной отчет, объединяющий несколько разных отчетов.

Характеристика генератора отчетов

В Delphi .NET 2006 для создания отчетов предназначен генератор отчетов Rave Reports 6.5.0 фирмы Nevrona, компоненты которого служат для управления отчетами. Генератор отчетов имеет развитые средства визуального проектирования отчетов. Компоненты, предназначенные для построения отчетов, находятся на странице **Rave** Палитры инструментов (рис. 17.1):

- `RvProject` (обеспечивает доступ к визуальным отчетам);
- `RvSystem` (обеспечивает просмотр, печать и настройку параметров отчетов);
- `RvNDRWriter` (обеспечивает хранение отчета в двоичном формате);
- `RvCustomConnection` (служит для настройки способа передачи данных в отчет);
- `RvDataSetConnection` (обеспечивает доступ к компонентам, производным от класса `TDataSet`);
- `RvTableConnection` (служит для доступа к компонентам типа `TTable` или их наследникам;

- `RvQueryConnection` (служит для доступа к компонентам типа `TQuery` или их потомкам);
- `RvRenderPreview` (посылает отчет в двоичном формате для предварительного просмотра);
- `RvRenderPrinter` (посылает отчет в двоичном формате на текущее устройство печати);
- `RvRenderPDF` (преобразует поток или файл NDR (Rave snapshot report file, файл снимка отчета Rave) в формат PDF);
- `RvRenderHTML` (преобразует поток или файл NDR в формат HTML);
- `RvRenderRTF` (преобразует поток или файл NDR в документ RTF);
- `RvRenderTEXT` (преобразует поток или файл NDR в текстовый документ).

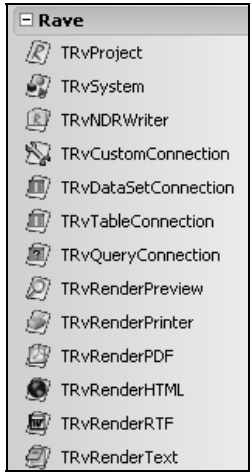


Рис. 17.1. Страница **Rave**

Указанная страница **Rave** присутствует в составе Палитры инструментов при разработке приложений VCL.NET.

Генератор отчетов Rave Reports позволяет создавать отчеты для обычных приложений и для приложений баз данных. При этом обеспечивается возможность создания отчетов для приложений баз данных, использующих различные механизмы доступа к данным.

Компоненты, расположенные на странице **Rave** Палитры инструментов, предназначены для управления отчетами в приложении. Важнейшими из них являются компоненты `RvProject` и `RvSystem`. Так, компонент `RvProject` типа `TRvProject` задает представление проекта отчета в приложении Delphi, а компонент `RvSystem` служит для управления отчетом и, к примеру, позволяет задать печать, предварительный просмотр или передачу отчета в файл.

Отчет размещается в файле с расширением rav, который представляет проект создаваемого отчета и содержит информацию об отчете, параметры оформления страниц отчета и др. В состав проекта отчета может входить один или несколько отчетов, в каждом из которых может быть произвольное число страниц и глобальных страниц. На страницах отчета и на глобальных страницах могут размещаться графические и текстовые объекты, связанные с источниками данных приложения.

Визуальный конструктор отчетов Rave Reports повышает удобство и упрощает непосредственную разработку отчетов.

Кроме того, в состав генератора отчетов входит ядро, которое обеспечивает управление отчетом, предварительный просмотр и отправку на печать. Код ядра при компиляции помещается в приложение, тем самым обеспечивается автономность последнего.

Визуальное конструирование отчетов

С помощью визуального конструктора отчетов Rave Reports можно быстро создать новый отчет, глобальную страницу, страницу отчета, настроить параметры отчета, выполнить подключение источника данных и др.

Интерфейс визуального конструктора

Вызов визуального конструктора отчетов можно выполнить с помощью команды **Tools | Rave Reports Designer** меню Delphi .NET 2006, двойным щелчком мыши на компоненте `RvProject` или с помощью команды **Rave Visual Designer** контекстного меню компонента `RvProject`. Возможный вид окна Rave Reports приведен на рис. 17.2.

Визуальный конструктор отчетов имеет практически стандартный интерфейс приложений Windows. В верхней части окна визуального конструктора расположено меню. Под ним находится панель инструментов, кнопки которой дублируют основные команды визуального конструктора. В правой верхней части конструктора размещена многостраничная панель инструментов, на вкладках которой содержатся компоненты, используемые при создании отчетов, а также элементы, управляющие параметрами объектов отчета (цвет линий, шрифт, выравнивание) и масштабом отображения отчета в окне.

Инспектор компонентов отчета, расположенный в левой части окна визуального конструктора, позволяет выполнить настройку свойств компонентов отчета, которые выбираются в дереве проекта отчета. В нижней части Инспектора компонентов отчета может отображаться подсказка для выбранного свойства. Управление отображением этой подсказки и подсветкой измененных свойств компонентов выполняется с помощью контекстного меню Инспектора компонентов отчета.

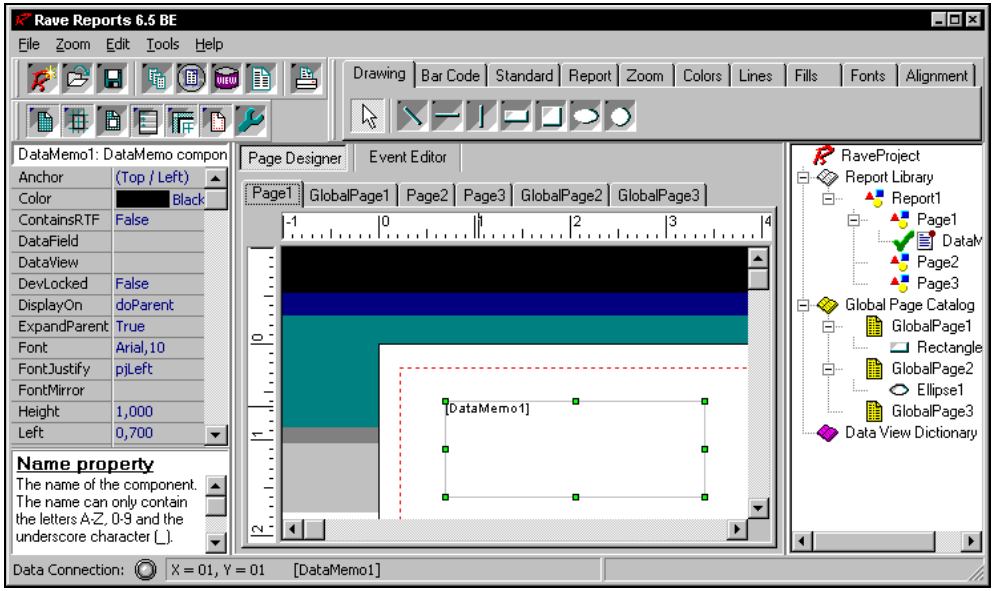


Рис. 17.2. Вид окна Rave Reports

В центральной части окна визуального конструктора расположен блокнот с двумя вкладками: **Page Designer** (Конструктор страниц) и **Event Editor** (Редактор событий). С помощью вкладки **Page Designer** можно добавлять, удалять и настраивать компоненты на отдельных страницах отчета. На этой вкладке имеется еще один блокнот, каждая из вкладок которого содержит одну страницу отчета. С помощью вкладки **Event Editor** создаются обработчики событий для отчетов, страниц, элементов оформления и компонентов отчета.

Дерево проекта отчета, размещенное в правой части окна визуального конструктора, отображает состав компонентов открытого проекта отчета. С помощью двойного щелчка удобно выбирать компоненты проекта отчета для отображения в центральной части окна и настройки свойств в Инспекторе компонентов.

Состав проекта отчетов

В дереве проекта отчетов Rave Project (см. рис. 17.2) имеются следующие три составляющие:

- **Report Library** (Библиотека отчетов проекта) — содержит все отчеты проекта;
- **Global Page Catalog** (Каталог глобальных страниц) — содержит перечень страниц, являющихся общими для всего проекта;
- **Data View Dictionary** (Словарь просмотров данных) — содержит объекты соединения с данными из внешних источников.

Рассмотрим более подробно каждую из названных составляющих проекта отчетов.

Библиотека отчетов проекта содержит все отчеты, входящие в состав проекта. Отдельный отчет в составе проекта включает произвольное число страниц, содержащих компоненты с данными, текстом и графикой, а также компоненты, повышающие наглядность оформления отчета.

При работе с проектом отчетов выделяют текущий отчет. При открытии проекта отчетов текущим становится первый отчет в составе проекта отчетов. При необходимости двойным щелчком мыши в дереве проекта отчетов можно установить текущим произвольный отчет. При выходе из среды визуального конструктора текущий отчет в составе проекта сохраняется и может использоваться при работе с отчетом с помощью компонента `RvProject` в приложении Delphi .NET 2006.

К проекту отчета можно добавлять новые отчеты с помощью команды **File** | **New Report** визуального конструктора. К текущему отчету можно добавлять страницы с помощью команды **File** | **New Report Page**. Для удаления отчета, страницы отчета или глобальной страницы достаточно выделить отчет или страницу отчета в дереве проекта отчетов и нажать клавишу <Delete>.

В отчет может входить произвольное число страниц. Перед предварительным просмотром и печатью отчета можно выбрать произвольное подмножество страниц отчета и добавить нужные страницы из каталога глобальных страниц. В отчете для управления составом и очередностью печати страниц предназначено свойство `PageList`. Если значение этого свойства не определено, то печатаются все страницы отчета в порядке их следования в дереве проектов отчета. Для задания значения свойства `PageList` используется редактор списка страниц **Page List Editor** (рис. 17.3), позволяющий сформировать список страниц для печати.

В проекте отчетов RaveDemo.rav, расположенном в папке \Borland\BDS\4.0RaveReports \Demos\Visual, имеется библиотека, содержащая множество различных отчетов. Например, в ее составе есть такие отчеты, как MultiPageReport (многостраничный отчет), BarCodes (штрихкоды), MasterDetailReport (отчет "мастер-детальный"). При необходимости полезно использовать отчеты из этой библиотеки в качестве шаблона при создании аналогичных отчетов.

Каталог глобальных страниц содержит страницы, которые являются доступными в любом отчете из состава библиотеки отчетов открытого проекта. Это позволяет получить стандартное оформление для ряда страниц отчета, таких как титульный лист, рамка для дипломных проектов и отчетов о НИР и т. п.

Напомним, что имеющуюся в каталоге глобальную страницу можно добавить в список страниц отчета для печати с помощью редактора списка страниц (см. рис. 17.3).

Для открытого проекта отчетов в состав каталога можно добавить пустую глобальную страницу, выполнив команду **File** | **New Global Page**. После этого глобальную страницу можно редактировать и добавлять в состав страниц любого из отчетов.

Словарь просмотров данных содержит объекты доступа к данным из внешних источников. Создание новых объектов выполняется после задания команды **File** |

New Data Object. При этом открывается диалоговое окно **Data Connections** (рис. 17.4), в котором для выбора предлагаются следующие типы объектов:

- **Data Lookup Security Controller** (Контроллер безопасности поиска данных) — обеспечивает аутентификацию пользователей по имени и паролю;
- **Database Connection** (Соединение с базой данных) — устанавливает соединение с внешним источником данных для требуемой технологии доступа (ADO, BDE, DBX, IBX);

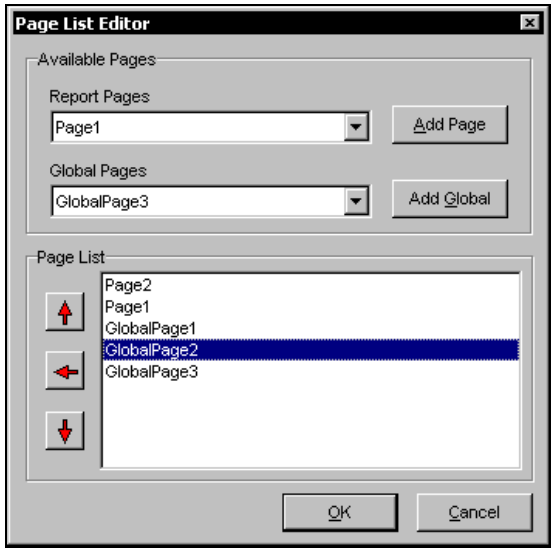


Рис. 17.3. Окно редактора Page List Editor

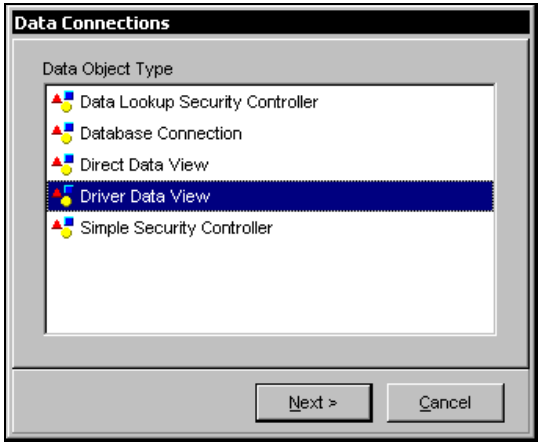


Рис. 17.4. Диалоговое окно **Data Connections**

- **Direct Data View** (Прямой просмотр данных) — создает просмотр данных для активного соединения с источником данных;
- **Driver Data View** (Просмотр данных с помощью драйвера) — создает просмотр данных на основе имеющегося в словаре соединения;
- **Simple Security Controller** (Простой контроллер безопасности) — задает список пользователей для возможной организации доступа в отчетах.

Объекты в составе словаря просмотров данных доступны для всех отчетов, входящих в состав проекта отчетов. Названные типы объектов используются преимущественно при создании отчетов для приложений баз данных.

Редактор событий

В проекте любому отчету, странице, элементу оформления или компоненту отчета можно назначить один или несколько обработчиков событий. Сделать это можно с помощью редактора событий, расположенного на вкладке **Event Editor** (рис. 17.5) блокнота в центральной части окна визуального конструктора.

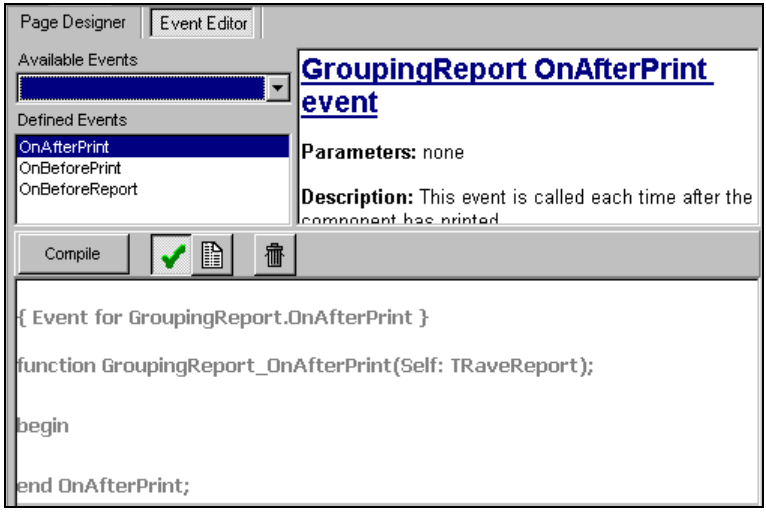


Рис. 17.5. Вкладка **Event Editor**

Создание обработчика событий выполняется следующим образом. В дереве проекта отчетов выбирается текущим отчет, страница или компонент, для которого требуется создать обработчик. Далее в списке **Available Events** выбирается одно из доступных для обработки событий. Это событие оказывается в списке **Defined Events** (Определенные события), автоматически создается его обработчик. Теперь для этого события следует задать собственно код обработки события в нижней части окна редактора событий и нажать кнопку **Compile**. При этом происходит

компиляция кода обработчика с выдачей сообщения о завершении компиляции или с сообщением об ошибке. При необходимости можно выбрать другое событие в списке **Available Events** и назначить ему свой обработчик.

При написании кода обработки событий используется синтаксис, близкий к синтаксису языка Object Pascal, допускается применение некоторых методов Delphi.

Компоненты многостраничной панели инструментов

Укажем назначение вкладок многостраничной панели инструментов, расположенной в верхней правой части окна визуального конструктора отчетов. Компоненты, используемые при создании отчетов, расположены на следующих вкладках:

- **Drawing** — графические компоненты, повышающие наглядность оформления отчета;
- **Bar Code** — компоненты, задающие различные варианты штрихкода;
- **Standard** — компоненты, задающие отображение текста и графики;
- **Report** — компоненты, предназначенные для отображения данных из внешних источников.

Элементы, управляющие параметрами объектов отчета (цвет линий, шрифт, выравнивание) и масштабом отображения отчета, расположены на следующих вкладках:

- **Zoom** — компоненты, управляющие масштабом отображения текущей страницы;
- **Colors** — компоненты, задающие цвета для графических компонентов и страниц;
- **Lines** — компоненты, задающие стиль и толщину линий графических компонентов;
- **Fills** — компоненты, задающие заливку графических компонентов;
- **Fonts** — компоненты, задающие параметры шрифта для текста;
- **Alignment** — компоненты, задающие выравнивание графических компонентов на странице.

Добавление компонентов в отчет выполняется традиционным для систем визуального программирования способом: щелчком мыши требуемый компонент выбирается на палитре и повторным щелчком мыши размещается в нужном месте страницы отчета. Дадим краткую характеристику компонентам, расположенным на различных страницах панели инструментов.

Страница **Drawing** многостраничной панели инструментов содержит компоненты, обеспечивающие создание графических объектов.

- `Line`, `Hline` и `Vline` — служат для рисования универсальной, горизонтальной и вертикальной линий соответственно. Все они имеют набор свойств, опреде-

ляющих длину и местоположение на странице отрезка линии. Для компонента `Line` отрезок можно развернуть произвольным образом визуально (с помощью мыши). Для двух других компонентов поворот отрезка также возможен, но не визуально, а только путем изменения свойств компонентов.

- `Rectangle` и `Square` — служат для рисования прямоугольника и квадрата, а компоненты `Ellipse` и `Circle` — для рисования эллипса и круга соответственно. Эти четыре компонента также имеют набор свойств, определяющих их размеры и местоположение на странице.

Страница **Bar Code** многостраничной панели инструментов содержит компоненты, задающие штрихкоды, которые реализуют различные стандарты.

- `PostNetBarCode` — соответствует коду PostNet почтовой службы США, задающему код адреса.
- `I2of5BarCode` — соответствует перемежающемуся коду I2of5 и служит для задания числовых последовательностей, в которых цифры поочередно кодируются штрихами и пробелами.
- `Code39BarCode` — соответствует коду Code39 и служит для кодирования цифр, заглавных латинских букв и ряда символов с помощью штрихов и пробелов.
- `Code128BarCode` — соответствует коду Code128 и служит для представления первых 128 символов кода ASCII.
- `UPCBarCode` — соответствует коду UPC (Universal Product Code) и служит для маркировки продуктов, может включать до 12 цифр.
- `EANBarCode` — соответствует коду EAN (European Article Numbering system), аналогичен коду UPC, может включать до 13 цифр.

Для всех компонентов кодируемое значение задается свойством `text`. Оно может быть также загружено из базы данных с помощью свойств `FileLink` и `DataLink`.

С помощью свойства `BarCodeRotation` штрихкод можно поворачивать с дискретностью 90°.

Страница **Standard** многостраничной панели инструментов содержит компоненты, задающие отображение текста и графики.

- `Text` — служит для отображения однострочного текста. Сам отображаемый текст задается путем редактирования в Инспекторе компонентов отчета значения свойства `text` этого компонента. Непосредственно на странице отчета отображаемый текст редактировать нельзя.
- `Memo` — служит для отображения многострочного текста. Отображаемый компонентом текст определяется свойством `text`, которое в Инспекторе компонентов отчета визуального конструктора связано с многострочным текстовым редактором Memo Editor.

  Остальные свойства компонентов `Text` и `Memo` позволяют настроить цвет, шрифт, выравнивание местоположения текста и др.

- `Section` (секция) — является невизуальным и предназначен для объединения в одну группу нескольких компонентов, к примеру, строк текста и изображений внутри заголовка отчета. Все размещенные внутри секции компоненты можно по отдельности перемещать в пределах секции, а при перемещении секции по странице вместе с секцией перемещаются и они. Выравнивание компонентов внутри секции осуществляется относительно ее границ.
- `Bitmap` — служит для отображения растровых изображений, хранящихся в файлах формата BMP. Отображаемый компонентом рисунок определяется с помощью свойства `Image`, которое в Инспекторе компонентов отчета визуального конструктора связано с диалогом загрузки файла изображения.
- `MetaFile` — служит для отображения изображений, хранящихся в файлах форматов EMF и WMF. Отображаемое компонентом изображение определяется с помощью свойства `Image`, которое в Инспекторе компонентов отчета визуального конструктора также связано с диалогом загрузки файла изображения. При этом файлы формата WMF таким способом загрузить нельзя, для этой цели нужно использовать свойство `FileLink` или `DataField`.

  Изображения, отображаемые с помощью компонентов `Bitmap` и `MetaFile`, можно масштабировать. При этом учитывается свойство `MatchSide`. При значениях этого свойства `msHeight` или `msWidth` (по умолчанию) высота или ширина изображения соответственно устанавливаются равными высоте (свойство `Height`) или ширине (свойство `Width`) исходной области, выделенной под изображение, а ширина или высота изображения соответственно устанавливаются с сохранением пропорций изображения. Если свойство `MatchSide` имеет значение `msBoth`, то ширина и высота изображения устанавливаются равными ширине и высоте выделенной области, при этом пропорции изображения могут быть нарушены. При значении `msInside` этого свойства изображение помещается внутри выделенной области с сохранением пропорций.
- `FontMaster` — является невизуальным и предназначен для задания единых параметров шрифта для группы компонентов отчета. С его свойством `Font` в Инспекторе компонентов отчета связано диалоговое окно определения параметров шрифта. Эти параметры шрифта служат для определения параметров шрифта всех компонентов отчета, в свойстве `FontMirror` которых указан данный компонент `FontMaster`.
- `PageNumInit` — является невизуальным и предназначен для задания нумерации страниц, начиная с той, где он расположен. Начальный номер при этом определяется свойством `InitValue`. При необходимости его значение можно загрузить из базы данных с помощью свойств `InitDataView` и `InitDataField`.

Компоненты отображения данных

Страница **Report** многостраничной панели инструментов визуального конструктора отчетов содержит компоненты, служащие для представления данных. Их

можно разделить на компоненты отображения данных из связанных с ними полей просмотра данных и компоненты управления структурой размещения данных (структурные компоненты), на которые помещаются компоненты отображения данных.

К компонентам *отображения данных* относятся следующие:

- `DataText` — служит для представления строковых или числовых значений полей связанного с ним просмотра данных;
- `DataMemo` — обеспечивает представление данных формата Memo или BLOB;
- `CalcText` — служит для представления результатов вычисления агрегатной функции по значениям связанного поля. Выполняемая функция выбирается из списка возможных значений свойства `CalcType`;
- `DataMirrorSection` — является невизуальным и подобно своему потомку `Section` служит для объединения других компонентов в одну группу.

У всех указанных компонентов представления данных свойство `DataView` задает связь с просмотром данных, а свойство `DataField` задает связь с полем просмотра, данные из которого отображаются компонентом.

К *структурным компонентам* относятся следующие:

- `Region` — служит для выделения области (части страницы отчета), на которой размещаются другие компоненты отображения данных. Более того, компоненты `Band` и `DataBand` размещаются только в области, определяемой компонентом `Region`. Свойство `Columns` этого компонента определяет число колонок при печати соответствующей части отчета;
- `Band` — задает полосу отчета, на которой располагаются другие компоненты отчета — отображения текста и графики, штрихкоды, отображения данных. Эта полоса служит для оформления заголовков, сносок, верхних и нижних колонтитулов и других фрагментов отчета, не изменяющиеся при печати просмотра данных;
- `DataBand` — задает полосу отчета, представляющую модель строки просмотра данных. На ней размещают компоненты отображения данных. При печати для каждой строки просмотра данных выводится новый экземпляр полосы со всеми расположенными на ней компонентами отчета. Тем самым в отчете построчно отображается весь просмотр данных.

У компонентов `Band` и `DataBand` имеется свойство, которое определяет стиль (назначение и поведение) полосы отчета. С этим свойством в Инспекторе компонентов отчета связан редактор стиля полосы **Band Style Editor**. В его окне (рис. 17.6) отображается взаимосвязь полос (компонентов `Band` и `DataBand`) в общей области (определяемой компонентом `Region`) и настраивается стиль текущей полосы.

В левой части окна редактора **Band Style Editor** отображается список полос области со взаимосвязями (реляционные отношения, группировка и вложенность и т. д.). Текущая полоса выделяется полужирным начертанием с подчеркиванием.

Полосы типа `Band` и `DataBand` верхнего уровня (т. е. не являющиеся подчиненными другим полосам типа `DataBand`) помечаются слева цветным ромбиком. При этом подчиненные полосы отчета типов `Band` и `DataBand` определяются с помощью их свойства `ControllerBand`, в качестве значения которого указывается имя полосы типа `DataBand`, которой они подчинены. Подчиненные полосы помечаются слева треугольником, острый угол которого направлен в сторону главной полосы, а цвет совпадает с цветом ромбика главной полосы.

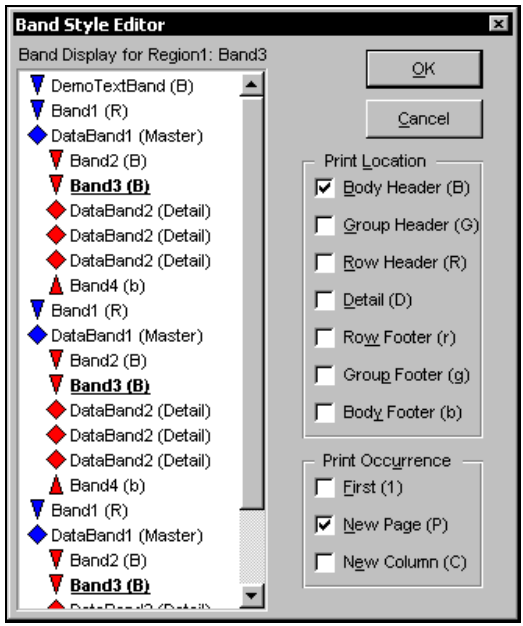


Рис. 17.6. Диалоговое окно **Band Style Editor**

В правой части окна редактора Band Style Editor с помощью группы флажков **Print Location** задается назначение текущей полосы отчета:

- **Body Header (B)** — заголовок в начале отчета;
- **Group Header (G)** — заголовок в начале группы, определенной в просмотре с помощью выражения `GROUP BY`;
- **Row Header (R)** — заголовок в начале каждой записи;
- **Detail (D)** — заголовок в начале подчиненного набора записей в отношении "один-ко-многим";
- **Row Footer (r)** — окончание записи;
- **Group Footer (g)** — окончание группы записей;
- **Body Footer (b)** — окончание отчета.

С помощью группы флажков **Print Occurrence** задается местоположение полосы в отчете:

- **First (1)** — на первой странице;
- **New Page (P)** — на новой странице;
- **New Column (C)** — в начале каждой колонки отчета.

Компоненты `DataCycle`, `CalcOp`, `CalcTotal` и `CalcController` являются невизуальными и предоставляют ряд дополнительных возможностей по вычислению данных на основе просмотров и отображению их в отчетах.

Компоненты управления отчетом

Компоненты, предназначенные для управления отчетами, находятся на странице **Rave** Палитры инструментов главного окна Delphi .NET 2006. С помощью этих компонентов можно выполнить все действия по настройке параметров, установлению соединения с источником данных, предварительному просмотру и печати имеющегося отчета с использованием генератора отчетов Rave Reports. Другое дело, что ряд этих действий, например предварительный просмотр и печать отчета, удобнее выполнить с использованием визуального конструктора отчетов Rave Reports.

Напомним, что страница **Rave** присутствует в составе Палитры инструментов только при разработке приложений VCL.NET.

Далее кратко освещены важнейшие компоненты управления отчетом, без которых практически невозможны предварительный просмотр и печать отчетов вне среды визуального конструктора отчетов.

Компонент-проект отчета

При подготовке и использовании отчета обязательным является использование компонента-проекта отчета `RvProject`. С помощью свойства `ProjectFile` типа `String` этого компонента в окне Инспектора компонентов отчета или программно устанавливается связь с проектом отчета (файлом проекта отчета с расширением rav). Например:

```
RvProject1.ProjectFile:='C:\Program Files\Borland\Delphi7\Rave5\ProjectSTR.rav';
```

Для обеспечения связи компонента `RvProject` с компонентом `RvSystem`, предназначенным для управления отчетом, служит свойство `Engine` типа `TRpComponent`. Естественно, перед заданием ссылки на компонент `RvSystem` последний должен быть предварительно помещен на форму приложения. Этот компонент может отсутствовать на форме приложения отчета, в этом случае при запуске отчета открывается диалоговое окно **Output Options**, в котором выбираются вариант дальнейших действий с отчетом (печать, просмотр или помещение в файл) и, при необходимости, формат файла.

В проекте отчета может быть несколько отчетов. Для их указания в компоненте `RvProject` предназначены свойства `ReportName`, `ReportFullName` и `ReportDesc` (соответственно задающие имя, полное имя и описание отчета), имеющие тип `String` и определяющие параметры текущего отчета. Для смены в проекте текущего отчета можно использовать функцию `SelectReport(ReportName: String; FullName: boolean): boolean`. Если параметр `FullName` имеет значение `True`, то параметр `ReportName` задает полное имя отчета, в противном случае — имя отчета.

Для получения списка имен отчетов, входящих в состав проекта, служит процедура `GetReportList(ReportList: TString; FullName: boolean)`. При ее вызове имена отчетов помещаются в список строк `ReportList`. При значении `True` параметра `FullName` в названный список помещаются полные имена отчетов, в противном случае — имена отчетов.

Для выполнения текущего отчета предназначены процедуры `Execute;` и `ExecuteReport(ReportName: String)`. Во втором случае имя отчета, задаваемое параметром процедуры, должно совпадать с именем отчета, определяемого свойством `ReportName` рассматриваемого нами компонента.

Компонент управления отчетом

Для управления отчетом служит компонент `RvSystem`, который обеспечивает выбор варианта действий с отчетом (печать, просмотр или помещение в файл) и при необходимости выбор формата файла. Как отмечалось, связь этого компонента с проектом отчета реализуется с помощью ссылки из компонента `RvProject` через свойство `Engine`.

Компонент `RvSystem` в своем составе инкапсулирует объекты, которые обеспечивают вывод отчета в один из трех системных приемников: `SystemPrinter` (печать отчета); `SystemPreview` (предварительный просмотр) и `SystemFiler` (вывод в файл).

Перед направлением отчета одному из системных приемников компонент `RvSystem` открывает диалоговое окно **Output Options**, в котором с помощью группы зависимых переключателей **Printer**, **Preview** и **File** осуществляется выбор системного приемника.

Компоненты установления соединения

Для установления соединения с источниками данных служат следующие компоненты страницы **Rave** Палитры компонентов:

- `RvCustomConnection` — соединение с источниками данных, не относящихся к базам данных (текстовые файлы, электронные таблицы и др.);
- `RvDataSetConnection` — соединение с наборами данных приложения баз данных;
- `RvTableConnection` — соединение с компонентом типа `TTable`, устанавливаемое при использовании механизма BDE.NET;

- `RvQueryConnection` — соединение с компонентом типа `TQuery`, устанавливаемое при использовании механизма BDE.NET.

При помещении на форму проекта приложения указанные компоненты оказываются доступными для выбора при создании просмотров **Direct Data View**. При этом предварительно компонент должен быть связан с соответствующим набором данных.

Для важнейшего (из упомянутых) компонента `RvDataSetConnection` свойство `DataSet` указывает на имя набора данных. Имя компонента `RvDataSetConnection`, задаваемое с помощью его свойства `Name`, используется в отчете для указания имени соединения в просмотрах данных отчета. Напомним, что используемое в просмотре имя соединения можно определить с помощью библиотеки просмотров **Data View Dictionary** как значение свойства `ConnectionName` для интересующего нас компонента `DataView`.

Схема управления отчетом и подсоединения данных

Мы рассмотрели назначение основных компонентов, служащих для управления отчетом и подсоединения данных к отчету. Отметим, что соединение с источниками данных в отчете можно выполнить двумя способами: с помощью драйвера Rave Reports, а также с помощью компонентов приложения доступа к данным и компонентов соединения. Для наглядности на рис. 17.7 приведена обобщенная схема управления отчетом и подсоединения данных к нему.

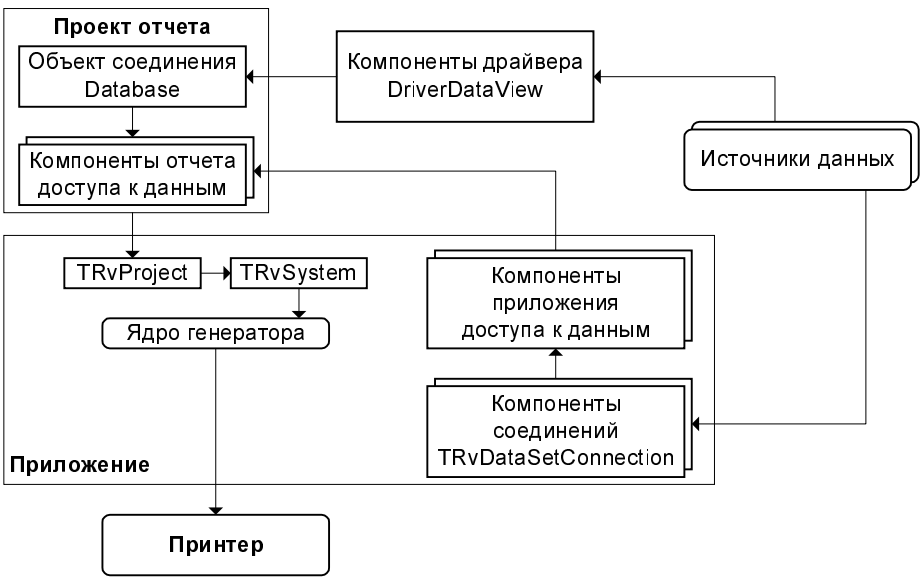


Рис. 17.7. Схема управления отчетом и подсоединения данных

Для организации соединения с помощью драйвера Rave Reports прежде всего командой **File | New Data Object** визуального конструктора отчетов создается объект соединения — компонент `Database`. При этом в открывшемся диалоговом окне **Data Connections** (см. рис. 17.4) выбирается вариант **Database Connection** и в очередном диалоговом окне нужно выбрать тип соединения (ADO, BDE, DBX или IBX). Каждому типу соединения соответствует свой драйвер (файл с расширением rdv). Далее в диалоговом окне **Database Connection Parameters** выполняется настройка параметров соединения, состав которых соответствует выбранному типу, и создается объект-соединение (с именем по умолчанию `DadabaseN`).

Теперь для связи компонентов данных отчета с данными соединения требуется создать просмотр данных, выполняющий формирование соответствующего набора данных. Для этого нужно командой **File | New Data Object** визуального конструктора открыть диалоговое окно **Data Connections** (см. рис. 17.4) и выбрать вариант **Driver Data View**. В очередном диалоговом окне требуется выбрать одно из имеющихся соединений проекта. При этом открывается диалоговое окно **Query Advanced Designer** (рис. 17.8), в котором на вкладке **Layout** следует перетащить нужные таблицы из выбранного нами соединения и задать связи между полями таблиц. Кнопка **Editor** открывает окно редактора с текстом сформированного запроса SQL, где его можно откорректировать вручную.

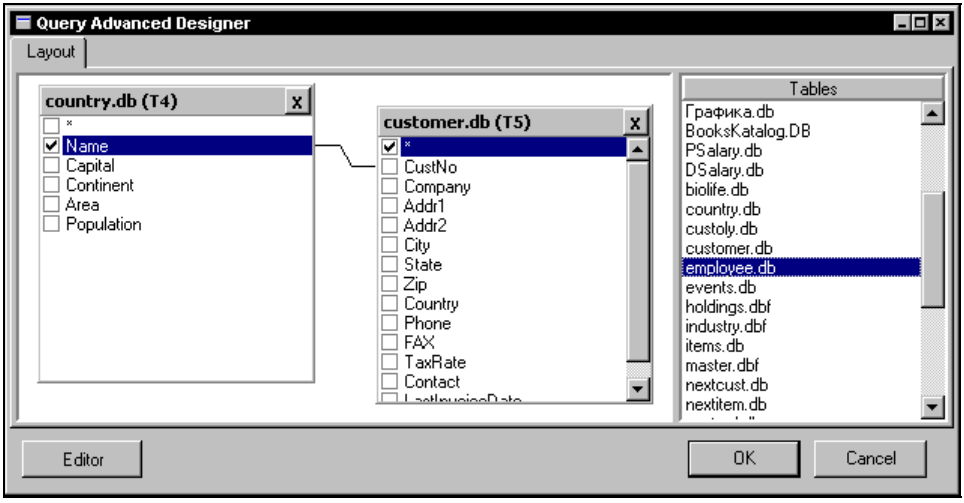


Рис. 17.8. Диалоговое окно **Query Advanced Designer**

После завершения настройки запроса SQL созданный нами объект-просмотр (компонент `DriverDataView`) появляется в словаре **Data View Dictionary** (с именем по умолчанию `DriverDataViewN`). Строка подготовленного таким образом запроса SQL является значением свойства `Query` объекта-просмотра. С его помощью можно повторно вызвать окно редактора **Query Advanced Designer**.

Теперь созданный нами объект-просмотр данных можно подключать к компонентам доступа к данным в отчете, таким как DataText, DataMemo и DataBand, с помощью их свойства DataView. Для таких компонентов, как DataText и DataMemo, с помощью свойства DataField выбирается имя поля из таблицы объекта-просмотра.

Организация соединения с помощью компонентов приложения доступа к данным и компонентов соединения выполняется следующим образом. Для каждого соединения в приложении предварительно создается компонент набора данных, например, типа TTable или TQuery, который соединяется с источником данных с помощью его свойств DatabaseName и TableName. Далее с таким компонентом набора данных устанавливается соединение с помощью соответствующего компонента: RvCustomConnection, RvDataSetConnection, RvTableConnection, RvQueryConnection страницы **Rave** Палитры компонентов.

Напомним, что для компонента RvDataSetConnection свойство DataSet указывает на имя набора данных. Имя компонента RvDataSetConnection, задаваемое с помощью его свойства Name, используется в отчете для указания имени соединения в просмотрах данных отчета. Как отмечалось, используемое в просмотре имя соединения можно определить с помощью библиотеки просмотров **Data View Dictionary** как значение свойства ConnectionName для компонента DataView.

Теперь в среде визуального конструктора отчетов требуется создать объект прямого просмотра, с помощью команды **File | New Data Object** открыв диалоговое окно **Data Connections** (см. рис. 17.4) и выбрав вариант **Direct Data View**. Затем в очередном диалоговом окне **Data Connections** нужно выбрать интересующее нас соединение.

Как и при предыдущем способе организации соединения, созданный таким образом объект просмотра данных — компонент DataView — можно подключать к компонентам доступа к данным в отчете, например, таким как DataText, DataMemo и DataBand.

Примеры создания и просмотра отчетов

Создание отчета включает три основных этапа:

1. Создание заготовки отчета в среде визуального конструктора.
2. Настройка соединения и подключение просмотра данных в случае создания отчета для приложения баз данных.
3. Размещение на форме приложения и настройка компонентов RvProject и RvSystem, которые осуществляют связь приложения с заготовкой отчета и управление предварительным просмотром и печатью отчета.

Заготовку собственно отчета в среде визуального конструктора можно выполнить так:

- путем настройки одного из уже имеющихся отчетов, например, в проекте RaveDemo.rav;
- путем конструирования отчета для обычного приложения или для приложения баз данных в среде визуального конструктора;
- для приложения баз данных можно создать простой табличный отчет или отчет для отношений типа "главный-подчиненный" с помощью Мастера отчетов.

В приведенном списке вариантов создания заготовки отчета второй вариант представляется наиболее трудоемким и требует хороших навыков работы с компонентами визуального конструктора.

Кроме перечисленных выше действий, при создании отчета можно также выполнять преобразование файлов отчета в различные форматы и др.

Предварительный просмотр отчета

Чтобы создать отчет путем настройки одного из уже имеющихся в проекте RaveDemo.rav (находится в папке \Borland\BDS\4.0RaveReports\Demos\Visual) отчетов, нужно выполнить предварительный просмотр представляющего интерес отчета, а затем уже заняться его настройкой.

Рассмотрим, как выполнить предварительный просмотр отчета `TwoDetails` из проекта RaveDemo.rav. Для отчетов приложений баз данных, использующих просмотры данных, к которым относится отчет `TwoDetails`, перед просмотром и выполнением нужно настроить соединения. Вид отчета `TwoDetails` в окне визуального конструктора на этапе проектирования и соответствующее ему дерево отчета приведены на рис. 17.9.

Чтобы определить имена просмотров данных, используемых компонентами отображения данных отчета (`DataText1`, `DataText2` и `DataText3`), поочередно выделим эти компоненты в окне проекта отчета и с помощью Инспектора компонентов отчета выясним значение свойства `DataView`. В нашем случае это просмотры с именами `CustomerDV`, `OrdersDV` и `ItemsDV`.

Далее нам нужно определить используемые этими просмотрами имена соединений. Для этого в словаре просмотров данных **Data View Dictionary** также поочередно выделим просмотры с полученными именами и по значению их свойств `ConnectionName` выясним имена соединений. В нашем случае это значения `CustomerCXN`, `OrdersCXN` и `ItemsCXN` соответственно.

Все это означает, что в приложении баз данных имеются три набора данных, с которыми мы должны установить связь с помощью трех компонентов `RvDataSetConnection` (размещенных на странице **Rave** Палитры компонентов) с именами `CustomerCXN`, `OrdersCXN` и `ItemsCXN`. Для связи компонентов `RvDataSetConnection` с наборами данных из базы данных используется свойство

`DataSet` этих компонентов. В качестве наборов данных в нашем примере используются таблицы из базы данных `DBDEMOS` с именами `orders.db`, `customer.db` и `items.db`.

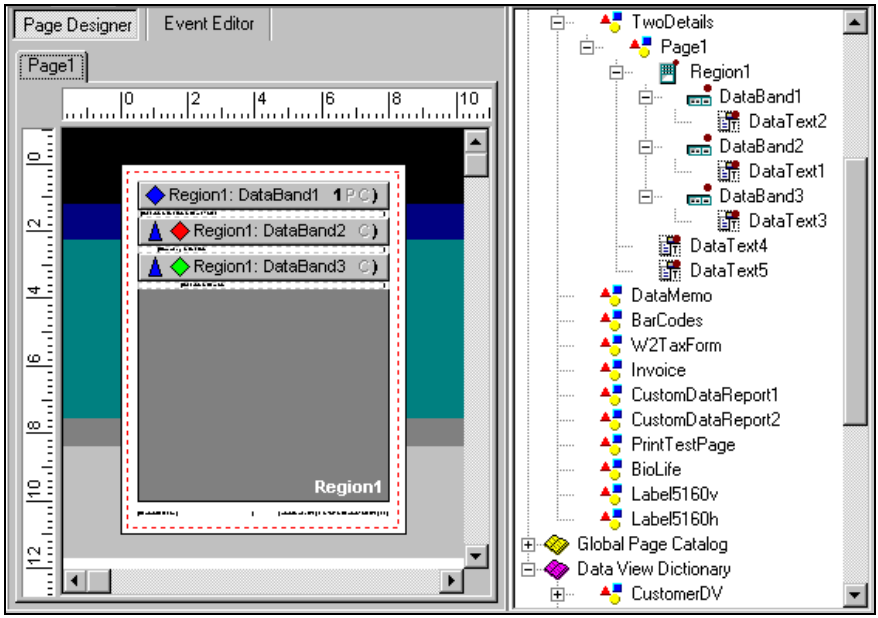


Рис. 17.9. Вид отчета `TwoDetails` в окне визуального конструктора

После настройки соединения отчета с базой данных приложения можно выполнить предварительный просмотр отчета `TwoDetails` в среде визуального конструктора, задав для этого команду **File | Execute Report**. Выберем в открывшемся диалоговом окне **Output Options** вариант **Preview** и нажмем кнопку **OK**. Вид отчета `TwoDetails` в окне предварительного просмотра приведен на рис. 17.10.

Из вида отчета на рис. 17.10 и из значений свойства `DataField` компонентов отображения данных `DataText1`, `DataText2` и `DataText3` видно, что в отчете компонент `DataText1` отображает данные из трех полей `OrderNO`, `CustomNO` и `ItemsTotal` таблицы `orders.db`; компонент `DataText2` отображает данные из полей `Company` и `CustNO` таблицы `customer.db`; компонент `DataText3` отображает данные из полей `OrderNO` и `ItemNO` таблицы `items.db`. Кроме того, компоненты `DataText4` и `DataText5` соответственно отображают номер текущей страницы из общего числа страниц и дату просмотра отчета.

Полученная нами при предварительном просмотре информация, а также данные о составе компонентов и их свойствах позволяют при необходимости настроить отчет соответствующим образом. Например, можно изменить таблицы базы данных приложения, можно изменить имена отображаемых полей и т. п.

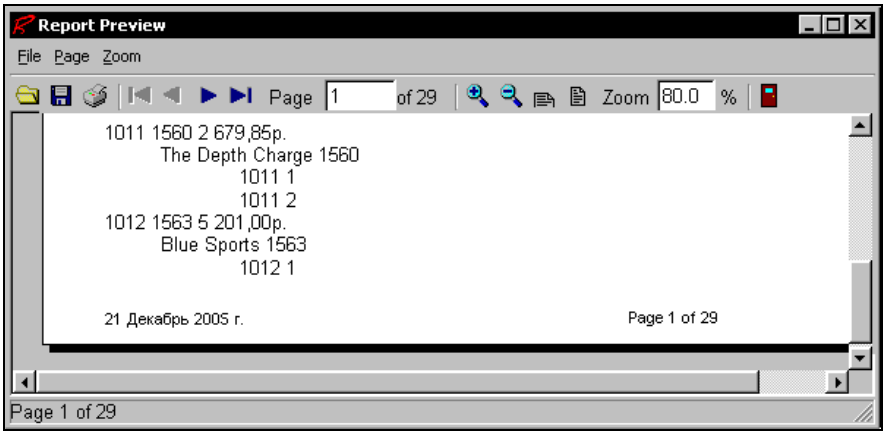


Рис. 17.10. Вид отчета `TwoDetails` при предварительном просмотре

Простой отчет приложения базы данных

Простой отчет представляет собой отчет на основе данных из одного набора данных и содержит сведения, которые выводятся в табличном виде без какой-либо дополнительной обработки данных (например, группирования). Размещение и вид печатаемых в отчете данных аналогичны размещению и виду данных, отображаемых в сетке `DBGrid`. Отличием является то, что данные отчета не размещаются на форме, а представлены в виде бумажного документа, и их нельзя редактировать.

Рассмотрим последовательность шагов, которую требуется выполнить при добавлении простого отчета к приложению базы данных.

1. Откроем приложение базы данных Delphi .NET, для которого требуется добавить простой отчет.
2. На форму приложения базы данных со страницы **Rave** Палитры поместим компонент `RvDataSetConnection` и с помощью Инспектора объектов присвоим свойству `DataSet` этого компонента значение, указывающее на имя набора данных, используемого в приложении.
3. С помощью визуального конструктора отчетов подготовим отчет и создадим файл проекта отчета. Для этого:
   - с помощью команды **Tools** | **Rave Reports Designer** меню Delphi .NET 2006 запустим визуальный конструктор Rave Reports;
   - выбором команды **File** | **New Data Object** откроем диалоговое окно **Data Connections** (см. рис. 17.4) и выберем в нем вариант **Direct Data View**;
   - в очередном диалоговом окне в списке **Active Data Connections** выберем вариант **RVDataSetConnection1** и нажмем кнопку **Finish**;

- в дереве проекта в правой части окна визуального конструктора отчетов раскроем узел **Data View Dictionary** и в нем раскроем вновь созданный узел **DataView1**;
- выбрав команду **Toolls** | **Report Wizards** | **Simple Table**, запустим Мастер создания простых таблиц в отчете, выберем вариант **DataView1** и нажмем кнопку **Next**;
- на последующих шагах работы с Мастером выберем поля таблицы для отображения в отчете, при необходимости изменим очередность следования полей, установим параметры полей страницы, текст заголовков и шрифтов, используемых в отчете (рис. 17.11);

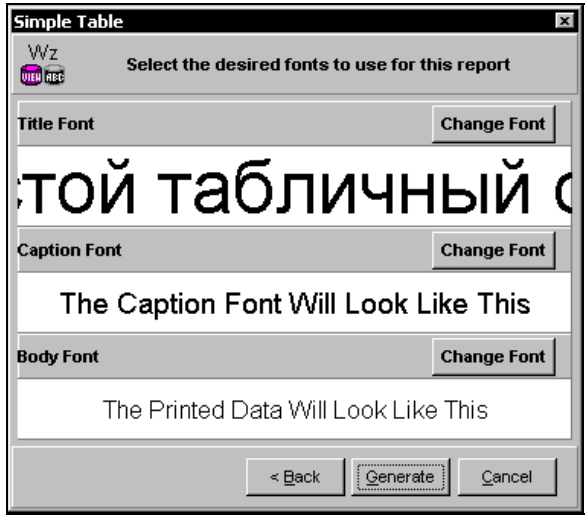


Рис. 17.11. Диалоговое окно настройки параметров шрифта

- на заключительном этапе работы с Мастером нажатием кнопки **Generate** запустим процесс генерации отчета;
- для просмотра сгенерированного отчета выберем команду **File** | **Execute Report**, в открывшемся диалоговом окне **Output Options** в поле **Report Destination** выберем переключатель **Preview** и нажмем **OK**. Вид полученного нами отчета при просмотре приведен на рис. 17.12;
- при необходимости выполним настройку параметров (шрифта, цвета и др.) отдельных составляющих созданного отчета;
- сохраним созданный нами проект отчета в файле с произвольным именем, например STRep, и расширением rav с помощью команды **File** | **Save As**;
- свернем или закроем диалоговое окно работы с визуальным конструктором отчета и вернемся к работе с приложением в Delphi .NET.

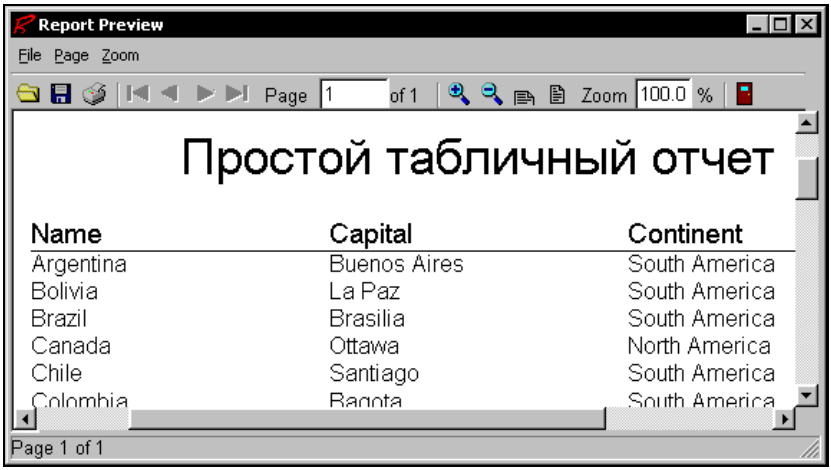


Рис. 17.12. Вид простого табличного отчета при просмотре

4. На форму приложения базы данных со страницы **Rave** Палитры поместим компонент проекта отчета `RvProject` и установим значение `C:\Program Files\Borland\BDS\4.0\RaveReports\STRep.RAV` (спецификация созданного нами файла проекта) свойству `ProjectFile` этого компонента с помощью Инспектора объектов.

5. На форму приложения базы данных со страницы **Standard** Палитры компонентов поместим кнопку (компонент `Button`) и в качестве обработчика события `OnClick` нажатия этой кнопки зададим вызов метода `ExecuteReport`, обеспечивающего выполнение некоторого отчета, например `MirrorReport` или `TwoDetails`, из состава проекта `STRep` отчета (компонента `RvProject1`) так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
RvProject1.Open;
try
  RvProject1.ExecuteReport('MirrorReport');;
finally
  RvProject1.Close;
end;
end;
```

6. С помощью клавиши <F9> запустим приложение на выполнение, и щелкнем мышью на кнопке формы, за которой закреплен созданный нами обработчик события. В открывшемся диалоговом окне выберем вариант печати отчета и нажмем кнопку **OK**.

В результате выполненных действий будет выдан на печать простой табличный отчет, содержащий данные из таблицы приложения базы данных, вид которого в окне просмотра приведен на рис. 17.12.

ЧАСТЬ V

Работа с базами данных VCL.NET

ГЛАВА 18

Компоненты приложения для работы с базами данных

Как отмечалось, одно- и двухуровневые приложения VCL.NET могут осуществлять доступ к локальным и удаленным БД с использованием следующих технологий: BDE.NET, dbGo.NET на основе ADO, dbExpress.NET и InterBase.NET.

Заметим, что приложения для работы с базами данных, подготовленные в среде Delphi версий до 7 включительно для технологий BDE, ADO, dbExpress и InterBase соответственно, могут быть легко перенесены в среду Delphi .NET 2006. Основным условием для этого является соблюдение ограничений языка Delphi .NET *(см. главу 2)*.

В этой главе мы укажем состав компонентов приложения для работы с базами данных VCL.NET. Затем рассмотрим компонент-источник данных, используемый для подключения наборов данных, и визуальные компоненты, которые служат для навигации по набору данных, отображения и редактирования записей. Эти два типа компонентов применяются при создании приложений для работы с базами данных VCL.NET с использованием любой из названных технологий доступа.

Характеристика приложения для работы с базами данных

При создании приложения для работы с базами данных VCL.NET с применением любой технологии доступа к данным используется стандартный состав и схема связи компонентов и таблицы базы данных. А именно, в состав приложения для работы с базами данных VCL.NET входят три типа компонентов:

- наборы данных;
- источник данных;
- визуальные компоненты.

Наборы данных служат для организации связи с таблицами базы данных. В Delphi .NET 2006 для разных механизмов доступа к данным в качестве наборов данных используются свои (но аналогичные) компоненты, такие как:

- `Table`, `Query`, `UpdateSQL`, `DecisionQuery` или `StoredProc` (для механизма BDE.NET);
- `ADOTable`, `ADOQuery` и `ADOStoredProc` (для механизма dbGo.NET на основе ADO);
- `SQLTable`, `SQLQuery` и `SQLStoredProc` (для механизма dbExpress.NET);
- `IBTable`, `IBQuery`, `IBUpdateSQL` и `IBStoredProc` (для механизма InterBease.NET).

Компоненты наборов данных мы рассмотрим отдельно в последующих главах.

Компонент-источник данных играет роль связующего звена между набором данных и визуальными компонентами. Визуальные компоненты служат для навигации по набору данных, отображения и редактирования записей.

Взаимосвязь компонентов приложения VCL.NET с таблицей, БД и используемые при этом свойства компонентов показаны на рис. 18.1.

Разрабатывая приложение, можно задавать значения всех свойств компонентов с помощью Инспектора объектов. При этом требуемые значения либо непосредственно вводятся в поле, либо выбираются в раскрывающихся списках. В последнем случае приложение создается с помощью мыши и не требует набора каких-либо символов на клавиатуре.

Для примера в табл. 18.1 приведены компоненты, используемые для доступа к данным таблицы `Clients.dbf` базы данных `dbdemos` с помощью механизма BDE.NET, их основные свойства и значения этих свойств. В качестве набора данных здесь используется компонент `Table1`.

В дальнейшем при организации приложений, использующих механизм доступа BDE.NET, предполагается, что названные компоненты связаны между собой именно таким образом, и свойства, с помощью которых эта связь осуществляется, не рассматриваются. Для приложений, использующих другие упоминавшиеся нами механизмы доступа, связь между компонентами устанавливается аналогично.

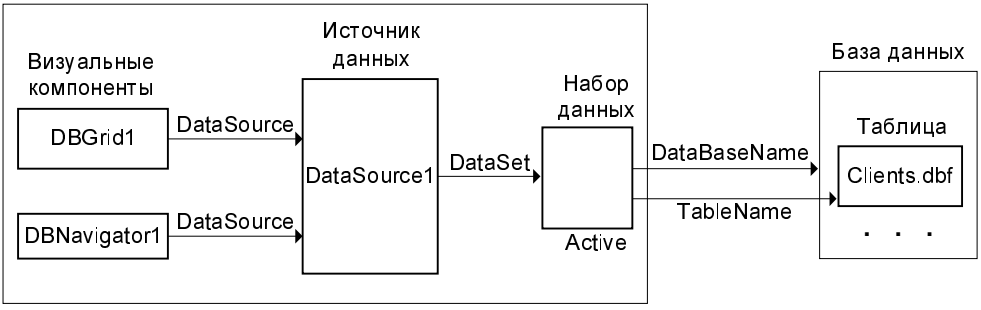


Рис. 18.1. Взаимосвязь компонентов приложения, таблицы и БД

Таблица 18.1. *Значения свойств компонентов*

Компонент	Свойства	Значения
Table1	DataBaseName	dbdemos
	TableName	Clients.dbf
	Active	True
DataSource1	DataSet	Table1
DBGrid1	DataSource	DataSource1
DBNavigator1	DataSource	DataSource1

Источник данных

Источник данных используется как промежуточное звено между набором данных и визуальными компонентами, с помощью которых пользователь управляет этим набором данных. В приложениях VCL.NET для работы с базами данных источник данных представлен компонентом `DataSource` типа `TDataSource`. Он размещен на странице **Data Access** Палитры инструментов.

Для указания набора данных, с которым связан источник данных, служит свойство `DataSet` типа `TDataSet` последнего. Визуальные компоненты связаны с источником данных через свои свойства `DataSource`. Обычно связь между источником и набором данных устанавливается на этапе проектирования в Инспекторе объектов, однако при необходимости эту связь можно установить или разорвать динамически. При смене у компонента `DataSource` набора данных визуальные компоненты автоматически подключаются к новому набору данных.

Указать набор данных можно, например, так:

```
DataSource2.DataSet := Nil;
DataSource1.DataSet := Table2;
```

Здесь для компонента `DataSource2` связь с набором данных разрывается, а для компонента `DataSource1` назначается набор данных `Table2`.

Для анализа состояния, в котором находится набор данных, можно использовать свойство `State` типа `TDataSetState`. При каждом изменении состояния набора данных для связанного с ним источника данных `DataSource` генерируется событие `OnStateChange` типа `TNotifyEvent`.

Если набор данных является редактируемым, то свойство `AutoEdit` типа `Boolean` определяет, может ли он автоматически переводиться в режим модификации при выполнении пользователем определенных действий. Например, для компонентов `DBGrid` и `DBEdit` таким действием является нажатие алфавитно-цифровой клавиши, когда компонент находится в фокусе ввода. По умолчанию свойство `AutoEdit`

имеет значение `True`, и автоматический переход в режим модификации разрешен. Если необходимо защитить данные от случайного изменения, то одной из предпринимаемых мер является установка свойству `AutoEdit` значения `False`.

Замечание

Свойства `AutoEdit` влияет на возможность редактирования набора данных только со стороны пользователя. Программно можно изменять записи независимо от значения этого свойства.

Без учета значения свойства `AutoEdit` пользователь может переводить набор данных в режим модификации путем нажатия кнопок компонента `DBNavigator`.

При изменении данных текущей записи генерируется событие `OnDataChange` типа `TDataChangeEvent`, описанного так:

```
type TDataChangeEvent = procedure (Sender: TObject; Field: TField) of object;
```

Параметр `Field` указывает на измененное поле; если данные изменены более чем в одном поле, то этот параметр имеет значение `Nil`. Следует иметь в виду, что в большинстве случаев событие `OnDataChange` генерируется и при переходе к другой записи. Это происходит, если хотя бы одно поле записи, ставшей текущей, содержит значение, отличное от значения этого же поля для записи, которая была текущей. Событие `OnDataChange` можно использовать, например, для контроля положения указателя текущей записи и выполнения действий, связанных с его перемещением. Это событие генерируется также при открытии набора данных.

Например, с помощью обработчика названного события контроль перемещения указателя текущей записи можно выполнить так:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
Label1.Caption := 'Запись номер ' + IntToStr(Table1.RecNo);
end;
```

При модификации текущей записи генерируется также 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.

Визуальные компоненты

Визуальные компоненты для работы с данными расположены на странице **Data Controls** Палитры инструментов (рис. 18.2). Они используются для навигации по набору данных, а также для отображения и редактирования записей. Часто эти компоненты называют *элементами, чувствительными к данным*.

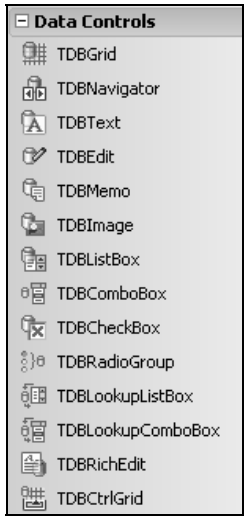


Рис. 18.2. Страница **Data Controls**

Одни визуальные компоненты для работы с данными предназначены для выполнения операций с полями отдельной записи, они отображают и позволяют редактировать значение поля текущей записи. К таким компонентам относятся, например, однострочный редактор `DBEdit` и графическое изображение `DBImage`.

Другие компоненты служат для отображения и редактирования сразу нескольких записей. Примерами таких компонентов являются сетки `DBGrid` и `DBCtrlGrid`, выводящие записи набора данных в табличном виде.

Визуальные компоненты для работы с данными похожи на соответствующие компоненты страниц **Standard**, **Additional** и **Win32** и отличаются, в основном, тем, что ориентированы на работу с БД и имеют дополнительные свойства `DataSource` и `DataField`. Первое из них указывает на источник данных — компонент `DataSource`, второе — на поле набора данных, с которым связан визуальный компонент. Например, однострочный редактор `DBEdit` работает так же, как однострочный редактор `Edit`, отображая строковое значение и позволяя пользователю изменять его. Отличие компонентов состоит в том, что в редакторе `DBEdit` отображается и изменяется значение определенного поля текущей записи набора данных.

Отметим, что для всех визуальных компонентов, предназначенных для отображения и редактирования значений полей, при изменении пользователем их содержимого набор данных автоматически переводится в режим редактирования. Произведенные с помощью этих компонентов изменения также автоматически сохраняются в связанных с ними полях при наступлении определенных событий, таких, например, как потеря фокуса визуальным компонентом или переход к другой записи набора данных.

При программном изменении содержимого этих визуальных компонентов набор данных не переводится в режим редактирования автоматически. Для этой цели в коде должен предварительно вызываться метод `Edit` набора данных. Чтобы сохранить изменения в поле (полях) текущей записи, программист также должен предусмотреть соответствующие действия, например, вызов метода `Post` или переход к другой записи набора данных.

В табл. 18.2 приводятся так называемые стандартные, дополнительные и 32-разрядные визуальные компоненты, расположенные на страницах **Standard**, **Additional** и **Win32** Палитры инструментов, а также соответствующие им визуальные компоненты для работы с данными в приложениях VCL.NET (страница **Data Controls**).

Таблица 18.2. *Соответствие визуальных компонентов для работы с данными*

Компоненты страниц *Standard*, *Additional* и *Win32*	**Компоненты страницы *Data Controls***
`Label`	`DBText`
`Edit`	`DBEdit`
`Memo`	`DBMemo`
`RichEdit`	`DBRichEdit`
`ListBox`	`DBListBox`
`ComboBox`	`DBComboBox`
`CheckBox`	`DBCheckBox`
`RadioGroup`	`DBRadioGroup`
`Image`	`DBImage`
`StringGrid`	`DBGrid`
`Chart`	`DBChart`

Рассмотрим особенности отдельных визуальных компонентов, предназначенных для работы с данными.

Отображение и редактирование значения строкового поля

Для отображения и редактирования значения строкового поля служит компонент `DBEdit` типа `TDBEdit`. Как отмечалось, однострочный редактор `DBEdit` работает так же, как однострочный редактор `Edit`, отображая строковое значение и позволяя пользователю изменять его. Отличие компонентов состоит в том, что в редакторе `DBEdit` отображается и изменяется значение определенного поля текущей записи набора данных.

Чтобы компонент DBEdit отображал и позволял редактировать значения некоторого поля набора данных, нужно установить его свойству DataSource в качестве значения имя источника данных, а свойству DataField — имя поля связанного с этим источником набора данных. Сделать это можно с помощью Инспектора объектов или программно. Например:

```
DBEdit1.DataSource := DataSource1;
DBEdit1.DataField := 'Name';
```

Отображение и редактирование значения логического поля

Логическое поле (поле логического типа) может содержать одно из двух значений: True (истина) или False (ложь). Разрешается использование прописных букв и сокращение вводимого значения, т. е. допустимы значения True, true, tru, Tr, t и т. д.

Для отображения и изменения значения логического поля можно использовать редактор DBEdit. Однако удобнее выполнять эти действия с помощью флажка (независимого переключателя) DBCheckBox, который позволяет "включить" или "выключить" значение логического поля.

Флажок DBCheckBox является аналогом компонента CheckBox, поэтому здесь мы остановимся только на свойствах, характерных именно для этого флажка.

Компонент DBCheckBox выглядит на экране как квадратик (флажок) с текстовым заголовком. Если в нем находится галочка (при этом говорят, что флажок "включен" или "установлен"), то связанное с этим флажком логическое поле текущей записи содержит значение True. Если же квадратик пуст (флажок снят), то логическое поле текущей записи содержит значение False.

Флажок DBCheckBox можно применять также для отображения и редактирования строковых полей, если воспользоваться свойствами ValueChecked и ValueUnChecked.

Свойство ValueChecked типа String содержит строковые значения, которые устанавливают связанный с этим полем флажок во включенное состояние. Отдельные значения разделяются точкой с запятой. В качестве значений допускаются любые алфавитно-цифровые символы, в том числе русские буквы. Регистр алфавитных символов не различается, т. е. значения ДА и да считаются одинаковыми. Например:

```
DBCheckBox1.ValueChecked := 'True;T;Yes;Y;Да;Д';
```

Свойство ValueUnChecked типа String содержит строковые значения, которые устанавливают связанный с этим полем флажок в выключенное состояние. Значения задаются таким же образом, как и для свойства ValueChecked:

```
DBCheckBox1.ValueUnChecked := 'False;F;No;N;Нет;Н';
```

Если поле не содержит ни одного из значений, указанных в свойствах `ValueChecked` и `ValueUnChecked`, то флажок устанавливается в неопределенное состояние.

Отметим, что, несмотря на наличие приведенных свойств, возможности флажка `DBCheckBox` по редактированию строковых полей намного меньше, чем у элемента `DBEdit`.

Представление записей с помощью сетки

Для вывода записей набора данных в табличном виде удобно использовать сетку, представленную в приложениях VCL.NET компонентом `DBGrid` типа `TDBGrid`. Внешний вид сетки соответствует внутренней структуре таблицы БД и набора данных, при этом строке сетки соответствует запись, а столбцу — поле.

С помощью сетки пользователь управляет набором данных, поля которого в ней отображаются. Для навигации по записям и их просмотра используются полосы прокрутки и клавиши перемещения курсора. Для перехода в режим редактирования поля достаточно установить на него курсор и нажать любую алфавитно-цифровую клавишу. Переход в режим вставки новой записи выполняется нажатием клавиши <Insert>, после чего можно заполнять поля. Вставка записи происходит в том месте, где находится указатель текущей записи. Изменения, сделанные при редактировании или добавлении записи, подтверждаются нажатием клавиши <Enter> или переходом к другой записи или отменяются нажатием клавиши <Esc>. Для удаления записи следует нажать комбинацию клавиш <Ctrl>+ +<Delete>.

Сетка `DBGrid` отображает все записи, имеющиеся в наборе данных, поэтому количество строк в ней изменять нельзя.

Основным свойством сетки является свойство `Columns` типа `TDBGridColumns`, которое представляет собой массив (коллекцию) объектов `Column` типа `TColumn`, описывающих отдельные столбцы сетки.

Свойство `SelectedIndex` типа `Integer` задает номер текущего столбца в массиве `Columns`, а свойство `SelectedField` указывает на объект типа `TField`, которому соответствует текущий столбец сетки.

Свойство `FieldCount` типа `Integer`, доступное во время выполнения программы, содержит число видимых столбцов сетки, а свойство `Fields [Index: Integer]` типа `TField` позволяет получить доступ к отдельным столбцам. Индекс определяет номер столбца в массиве столбцов и принимает значения в интервале `0 .. FieldCount-1`.

Свойства `Color` и `FixedColor` типа `TColor` задают цвета сетки и ее фиксированных элементов соответственно. По умолчанию свойство `Color` имеет значение

clWindow (цвет фона Windows), а свойство FixedColor — значение clBtnFace (цвет кнопки).

Свойство TitleFont типа TFont определяет шрифт, используемый для вывода заголовков столбцов.

Доступ к параметрам сетки (например, для настройки) возможен через свойство Options типа TGridOptions. Это свойство представляет собой множество и принимает комбинации следующих значений:

- dgEditing (пользователю разрешается редактирование данных в ячейках);
- dgAlwaysShowEditor (сетка не блокирует режим редактирования);
- dgTitles (отображаются заголовки столбцов);
- dgIndicator (для текущей записи в начале строки выводится указатель);
- dgColumnResize (пользователь может с помощью мыши изменять размер столбцов и перемещать их);
- dgColLins (между столбцами выводятся разделительные вертикальные линии);
- dgRowLines (между строками выводятся разделительные горизонтальные линии);
- dgTabs (для перемещения по сетке можно использовать клавишу <Tab> и комбинацию клавиш <Shift>+<Tab>);
- dgRowSelect (пользователь может выделить целую строку); при установке этого значения игнорируются значения dgEditing и dgAlwaysShowEditor;
- dgAlwaysShowSelection (ячейка остается выделенной, даже если сетка теряет фокус);
- dgConfirmDelete (при удалении строки выдается запрос на подтверждение операции);
- dgCancelOnExit (добавленные к сетке пустые строки (записи) при потере сеткой фокуса не сохраняются в наборе данных);
- dgMultiSelect (в сетке можно одновременно выделить несколько строк).

По умолчанию свойство Options содержит комбинацию значений [dgEditing, dgTitles, dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs, dgConfirmDelete, dgCancelOnExit].

При щелчке на ячейке с данными генерируется событие OnCellClick, а щелчок на заголовке столбца вызывает событие OnTitleClick. Оба события имеют тип TDBGridClickEvent, описываемый так:

```
type TDBGridClickEvent = procedure (Column: TColumn) of object;
```

Параметр Column представляет собой столбец, на котором был произведен щелчок.

При перемещении фокуса между столбцами сетки инициируются события OnColEnter и OnColExit типа TNotifyEvent, первое из которых возникает при получении столбцом фокуса, а второе — при его потере.

Если свойство `Options` содержит значение `dgColumnResize`, то пользователь может с помощью мыши перемещать столбцы сетки. При таком перемещении генерируется событие `OnColumnMoved` типа `TMovedEvent`, описываемого как:

```
type TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex:
Longint) of object;
```

Параметры `FromIndex` и `ToIndex` указывают индексы в массиве столбцов сетки, соответствующие предыдущему и новому положению перемещенного столбца соответственно.

Сетка `DBGrid` способна автоматически отображать в своих ячейках информацию, но при необходимости программист может выполнить и собственное отображение сетки. Это может понадобиться в случае, когда желательно выделить ячейку или столбец с помощью цвета или шрифта, а также вывести в ячейке, кроме текстовой информации, и графическую информацию, например, небольшой рисунок. Для программной реализации отображения сетки используется обработчик события `OnDrawColumnCell` типа `TDrawColumnCellEvent`, которое возникает при прорисовке любой ячейки.

Столбцы сетки

Отдельный столбец `Column` сетки представляет собой объект типа `TColumn`. По умолчанию для каждого поля набора данных, связанного с компонентом `DBGrid`, автоматически создается отдельный столбец, и все столбцы в сетке доступны. Такие столбцы являются *динамическими*. Для создания *статических* столбцов используется специальный Редактор столбцов. Если хотя бы один столбец сетки является статическим, то динамические столбцы уже не создаются ни для одного из оставшихся полей набора данных. Причем в наборе данных доступными являются статические столбцы, а остальные столбцы считаются отсутствующими. Изменить состав статических столбцов можно с помощью Редактора столбцов на этапе разработки приложения.

Взаимодействие между динамическими и статическими столбцами, а также Редактором столбцов аналогично взаимодействию между динамическими и статическими полями набора данных и Редактором полей.

Характеристики и поведение сетки и ее отдельных столбцов во многом определяются полями набора данных (а также соответствующими объектами типа `TField`), для которых создаются объекты типа `TColumn`.

Функционирование динамических столбцов зависит от свойств объекта поля: при изменении свойств объекта типа `TField` соответственно изменяются свойства объекта типа `TColumn`. К примеру, динамический столбец получает от поля такие характеристики, как имя и ширину.

Достоинством статических столбцов является то, что для их объектов можно установить значения свойств, отличные от свойств соответствующего поля и не

зависящие от него. Например, если для некоторого статического столбца установить свое имя, то оно не будет меняться даже в случае, если с этим столбцом связывается другое поле набора данных. Кроме того, объекты типа `TColumn` статических столбцов создаются на этапе разработки приложения, и их свойства доступны с помощью Инспектора объектов.

Для запуска Редактора столбцов (рис. 18.3) можно вызвать контекстное меню компонента `DBGrid` и выбрать в нем пункт **Columns Editor**. Редактор столбцов можно вызвать также щелчком мыши на свойстве `Columns` в окне Инспектора объектов.

В заголовке Редактора столбцов выводится составное имя массива столбцов, например, `DBGrid1.Columns`. Под заголовком находится панель инструментов, видимостью которой можно управлять с помощью пункта **Toolbar** контекстного меню Редактора столбцов. Большую часть Редактора столбцов занимает список статических столбцов, при этом столбцы перечисляются в порядке их создания (этот порядок может отличаться от исходного порядка полей в наборе данных).

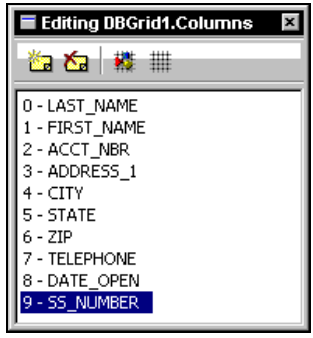


Рис. 18.3. Диалоговое окно Редактора столбцов

Замечание

При изменении порядка столбцов сетки автоматически изменяется порядок связанных с ними полей набора данных, что необходимо учитывать при доступе к полям по номерам объектов типа `TField`, а не по именам объектов.

Первоначально список статических столбцов пуст, показывая тем самым, что все столбцы сетки являются динамическими. Редактор столбцов позволяет:

- создать статический столбец;
- удалить статический столбец;
- изменить порядок следования статических столбцов.

Для любого выбранного в Редакторе статического столбца (объекта типа `TColumn`) через Инспектор объектов можно задать или изменить его свойства и определить

обработчики его событий. Это допустимо потому, что соответствующие статическим столбцам объекты типа `TColumn` доступны уже на этапе разработки приложения.

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 (т. е. столбца, выбранного в списке Редактора столбцов) доступны с помощью Инспектора объектов.

Перечислим наиболее важные свойства объекта столбца.

- `Alignment` типа `TAlignment` — управляет выравниванием значений в ячейках столбца и может принимать следующие значения:
  - `taLeftJustify` (выравнивание по левой границе);
  - `taCenter` (выравнивание по центру);
  - `taRightJustify` (выравнивание по правой границе).
- `Count` типа `Integer` — указывает число столбцов сетки.
- `Field` типа `TField` — определяет объект поля набора данных, связанный со столбцом.
- `FieldName` типа `String` — указывает имя поля набора данных, с которым связан столбец. При установке этого свойства с помощью Инспектора объектов значение можно выбирать в списке.
- `PickList` типа `TStrings` — представляет собой список для выбора заносимых в поле значений. Текущая ячейка совместно со списком `PickList` образует своего рода компонент `ComboBox` или `DBComboBox`. Если для столбца сформирован список выбора, то при попытке редактирования ячейки этого столбца справа появляется стрелка, при нажатии которой список раскрывается и позволяет выбрать одно из значений. При этом можно ввести в ячейку любое допустимое значение.
- `Title` типа `TColumnTitle` — представляет собой объект заголовка столбца. В свою очередь этот объект имеет такие свойства, как `Caption`, `Alignment`, `Color` и `Font`, определяющие название, выравнивание, цвет и шрифт заголовка соответственно.

Свойства столбца, управляющие форматированием, видимостью или возможностью модификации значений, не отличаются от соответствующих свойств поля.

Рассмотрим пример по установке свойств 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 с помощью Инспектора объектов
procedure TForm1.FormCreate(Sender: TObject);
begin
// Установка параметров второго столбца
DBGrid1.Columns[1].Title.Caption := 'Количество';
DBGrid1.Columns[1].Title.Alignment := taCenter;
DBGrid1.Columns[1].Alignment := taCenter;
// Установка параметров третьего столбца
DBGrid1.Columns[2].Title.Alignment := taCenter;
DBGrid1.Columns[2].Alignment := taLeftJustify;
DBGrid1.Columns[2].PickList.Clear;
DBGrid1.Columns[2].PickList.Add('Кредит');
DBGrid1.Columns[2].PickList.Add('Наличными');
DBGrid1.Columns[2].PickList.Add('Безнал');
// Установка параметров четвертого столбца
DBGrid1.Columns[3].Title.Alignment := taCenter;
DBGrid1.Columns[3].Alignment := taRightJustify;
// Скрытие пятого столбца
DBGrid1.Columns[4].Visible := false;
end;
```

Для второго столбца задается его заголовок, для второго, третьего и четвертого столбца задается выравнивание заголовка и значений. Пятый столбец устанавливается невидим. Кроме того, для третьего столбца создан список выбора. Установка свойств столбцов произведена с помощью обработки события создания формы, эти же действия можно выполнить с помощью Инспектора объектов. При выполнении приложения форма имеет вид, показанный на рис. 18.4.

Рис. 18.4. Задание свойств для столбцов сетки

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.

Рассмотрим пример использования компонента `DBGrid`, в котором осуществляется преобразование значений записей набора данных в текст.

В качестве набора данных используется компонент `Table1`, записи которого отображаются в сетке `DBGrid1`. При нажатии кнопки **Преобразовать в текст** (`Button1`) происходит последовательный просмотр полей всех записей набора данных (сетки) и преобразование их в текст, который помещается в многострочное поле редактирования `Memo1` (рис. 18.5).

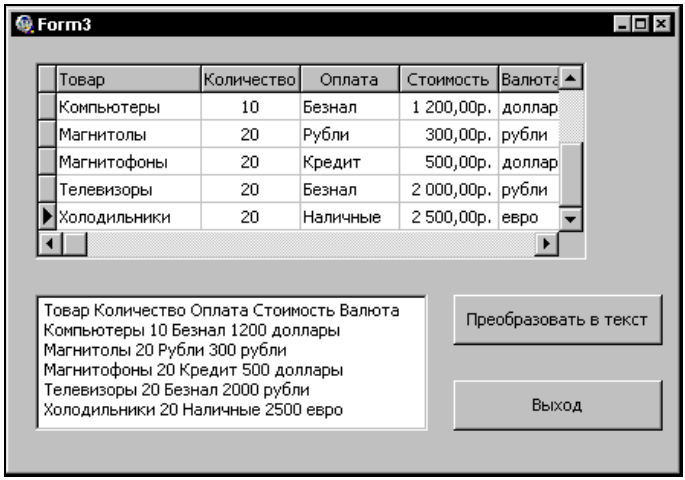


Рис. 18.5. Преобразование записей в текст

Далее приведен код обработчика события нажатия кнопки **Преобразовать в текст** (`Button1`).

```
procedure TForm3.Button1Click(Sender: TObject);
var i, n: integer;
    s, rs: string;
begin
               Memo1.Clear;
               Table1.First;
       // Перебор всех записей набора данных
for n := 1 to Table1.RecordCount do
begin
  rs := ''; s := '';
       // Чтение названий столбцов сетки
 if n = 1 then
 begin
         for i := 0 to DBGrid1.Columns.Count — 1 do
     begin
          s := DBGrid1.Columns[i].FieldName + ' ';
               rs := rs + s;
          end;
```

```
        Memo1.Lines.Add(rs);
        rs := ''; s := '';
 end;
              // Чтение значений полей текущей записи
              for i := 0 to DBGrid1.Columns.Count — 1 do
      begin
                s := DBGrid1.Columns[i].Field.AsString + ' ';
                rs := rs + s;
                end;
              Memo1.Lines.Add(rs);
              Table1.Next;
end;
end;
```

Для доступа к названиям и значениям полей набора данных использованы свойства `FieldName`, `Count` и `Field` столбцов сетки.

Использование навигатора

Для управления набором данных можно использовать навигатор, который обеспечивает соответствующий интерфейс пользователя. По внешнему виду и организации работы навигатор похож на мультимедийный проигрыватель. В приложениях VCL.NET навигатор представлен компонентом `DBNavigator` (рис. 18.6).

Рис. 18.6. Навигатор

Навигатор содержит кнопки, обеспечивающие выполнение различных операций с набором данных путем автоматического вызова соответствующих методов. *Состав видимых кнопок* определяет свойство `VisibleButtons` типа `TButtonSet`, принимающее комбинации следующих значений (в скобках указан вызываемый метод):

- `nbFirst` — перейти к первой записи (`First`);
- `nbPrior` — перейти к предыдущей записи (`Prior`);
- `nbNext` — перейти к следующей записи (`Next`);
- `nbLast` — перейти к последней записи (`Last`);
- `nbInsert` — вставить новую запись (`Insert`);
- `nbDelete` — удалить текущую запись (`Delete`);
- `nbEdit` — редактировать текущую запись (`Edit`);
- `nbPost` — утвердить результат изменения записи (`Post`);

- nbCancel — отменить изменения в текущей записи (Cancel);
- nbRefresh — обновить информацию в наборе данных (Refresh).

По умолчанию в навигаторе видимы все кнопки.

Метод BtnClick(Index: TNavigateBtn) служит для имитации нажатия кнопки, заданной параметром Index. Тип TNavigateBtn этого параметра идентичен типу TButtonSet, возможные значения соответствующего параметра которого перечислены выше. В качестве примера приведем строку кода:

```
DBNavigator.BtnClick(nbNext);
```

В ней имитируется нажатие кнопки nbNext, вызывающей переход к следующей записи набора данных.

При нажатии кнопки nbDelete может появляться диалоговое окно, в котором пользователь должен подтвердить или отменить удаление текущей записи. Появлением *окна подтверждения* управляет свойство ConfirmDelete типа Boolean, по умолчанию имеющее значение True, т. е. окно подтверждения выводится. Если при отладке приложения установить это свойство в значение False, то запись будет удаляться без запроса подтверждения.

Свойство Flat типа Boolean управляет *внешним видом* кнопок. По умолчанию оно имеет значение False, и кнопки отображаются объемными. При установке свойства Flat в значение True кнопки приобретают плоский вид, соответствующий современному стилю.

Подсказку для отдельной кнопки можно установить с помощью свойства Hints типа TString. По умолчанию список подсказок содержит текст на английском языке, который можно заменить на русский, вызвав Строковый редактор (String list editor). Подсказка для навигатора устанавливается через свойство Hint типа String. Напомним, что для отображения подсказок нужно присвоить значение True свойству ShowHint, по умолчанию, имеющему значение False.

На практике часто вместо навигатора используются отдельные кнопки Button или BitBtn, при нажатии которых вызываются соответствующие методы управления набором данных. Например, в процедуре

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Table1.Next;
end;
```

при нажатии кнопки Button1 выполняется переход к следующей записи набора данных Table1.

ГЛАВА 19

Технология BDE.NET

Как отмечалось в предыдущей главе, в составе любого приложения для работы с базами данных с использованием VCL.NET важнейшую роль играют наборы данных. Как отмечалось, в Delphi .NET 2006 для работы с наборами данных при использовании механизма BDE.NET служат такие компоненты, как `Table`, `Query`, `UpdateSQL`, `DecisionQuery` или `StoredProc`. В случае других механизмов доступа для работы с наборами данных служат аналогичные компоненты: `ADOTable`, `ADOQuery` и `ADOStoredProc` (для механизма dbGo.NET на основе ADO), `SQLTable`, `SQLQuery` и `SQLStoredProc` (для механизма dbExpress.NET), `IBTable`, `IBQuery`, `IBUpdateSQL` и `IBStoredProc` (для механизма InterBease.NET).

В этой главе мы рассмотрим основные наборы данных, используемые для доступа к данным с помощью механизма BDE.NET. Кроме того, мы рассмотрим также технологии выполнения операций с данными: сортировка, навигация, фильтрация, поиск записей и модификация набора данных.

Наборы данных

Таблицы БД располагаются на диске и являются физическими объектами. Для операций с данными, содержащимися в таблицах, используются наборы данных. Компоненты, представляющие наборы данных, являются невизуальными.

Общая характеристика

В терминах системы Delphi .NET *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

Многие СУБД вместо термина *набор данных* используют термины *выборка* или *таблица*.

Наиболее универсальными и, соответственно, часто используемыми являются компоненты `Table` и `Query`, задающие наборы данных. Эти компоненты являются производными от класса `TDBDataSet` — потомка класса `TDataSet` (через класс `TBDEDataSet`). Они демонстрируют схожие с базовыми классами характеристики и поведение, но каждый из них имеет и свои особенности. Здесь мы рассмотрим наиболее общие характеристики наборов данных.

Расположение БД, с таблицами которой выполняются операции, указывает свойство `DatabaseName` типа `String`. Значением свойства является имя каталога, в котором находится БД (файлы ее таблиц), или псевдоним, ссылающийся на этот каталог. Если для БД определен псевдоним, то его можно выбрать в раскрывающемся списке окна Инспектора объектов.

Замечание

Задание имени БД через псевдоним облегчает перенос приложения и файлов БД в другие каталоги и на другие компьютеры, т. к. для обеспечения работоспособности приложения после изменения расположения БД достаточно изменить название каталога, на который ссылается псевдоним БД.

Для компонента `Table` использование свойства `DatabaseName` является единственной возможностью задать местонахождение таблиц БД. Для компонента `Query` дополнительно можно указать в запросе SQL путь доступа к каждой таблице.

Замечание

При задании расположения БД программным способом набор данных предварительно необходимо закрыть, установив его свойству `Active` значение `False`. В противном случае генерируется исключение.

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`).

Например, в обработчике события нажатия кнопки `Button1` производится перебор всех записей набора данных:

```
procedure TForm3.Button1Click(Sender: TObject);
var i: integer;
begin
Table1.First;
for i := 1 to Table1.RecordCount do begin
 // Инструкции по обработке очередной записи
 Listbox1.Items.Add(Table1.Fields.Fields[1].AsString);
 Table1.Next;
end;
end;
```

Здесь 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`.

В качестве инструкции обработки очередной записи используется заполнение списка (компонент `ListBox1`) значениями второго поля таблицы (`Table1.Fields.Fields[1]`) (рис. 19.1).

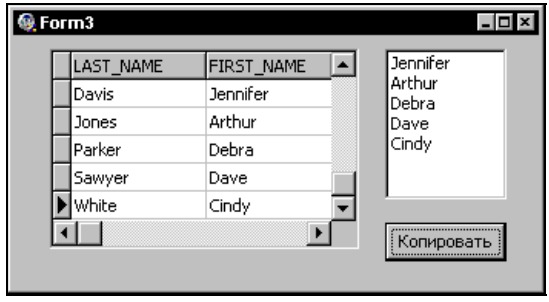


Рис. 19.1. Вид формы приложения после обработки записей таблицы

Для локальных таблиц dBase или Paradox составляющие набор данных записи последовательно нумеруются, отсчет начинается с единицы. Номер записи в наборе данных определяет свойство `RecNo` типа `Longint`, которое доступно во время выполнения программы.

Номер текущей записи можно узнать, например, так:

```
Edit1.Text := IntToStr(Table1.RecNo);
```

Замечание

При изменении порядка записей при сортировке или фильтрации нумерация записей также изменяется.

Для таблиц Paradox свойство `RecNo` можно использовать для перехода к требуемой записи, установив в качестве значения свойства номер записи.

Так, в операции

```
Table1.RecNo := StrToInt(Edit1.Text);
```

выполняется переход к записи, номер которой содержится в поле ввода `Edit1`. Указанная запись становится текущей.

Для выполнения операций с наборами данных используются два способа доступа к данным:

- навигационный;
- реляционный (SQL-ориентированный).

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 VCL.NET навигационный способ доступа к данным можно реализовать, используя любой из компонентов `Table` или `Query`.

Реляционный способ доступа основан на обработке *группы записей*. Если требуется обработать одну запись, все равно обрабатывается группа, состоящая из одной записи. При реляционном способе доступа используются SQL-запросы, поэтому его называют также *SQL-ориентированным*. Реляционный способ доступа ориентирован на работу с удаленными БД и является для них предпочтительным. Однако его можно использовать и для локальных БД. Реляционный способ доступа к данным в приложении можно реализовать с помощью компонента `Query`.

Состояния и режимы

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 открыть набор 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 свойства TableName или SQL.

Рассмотрим пример управления состоянием набора данных с помощью свойства Active, которое используется для открытия и закрытия набора данных Table1:

```
procedure TForm3.Button1Click(Sender: TObject);
begin
  Table1.Active:=False;
  Table1.DatabaseName:='DBDEMOS';
  Table1.TableName:='Animals.dbf';
  Table1.Active:=True;
end;
```

Если набор данных Table1 связан соответствующим образом с компонентами DataSource1 и DBGrid1, то при выполнении приложения нажатие кнопки приведет к отбору всех записей из таблицы Animals.dbf базы данных DBDEMOS и отображению их с помощью компонента DBGrid1.

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 так:

```
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.

Закрытие набора данных автоматически не сохраняет текущую запись, т. е. если набор данных при закрытии находился в режимах редактирования или вставки, то произведенные изменения данных в текущей записи будут потеряны. Поэтому перед закрытием набора данных нужно проверять его режим и при необходимости принудительно вызывать метод Post, сохраняющий сделанные изменения.

Одним из вариантов сохранения изменений является вызов метода `Post` в обработчике события `BeforeClose`, возникающего непосредственно перед закрытием набора данных.

Рассмотрим следующий пример:

```
procedure TForm1.Table1BeforeClose(DataSet: TDataSet);
begin
if (Table1.State = dsEdit) or (Table1.State = dsInsert) then Table1.Post;
end;
```

Если набор данных `Table1` находится в режиме редактирования или вставки, то перед его закрытием внесенные изменения сохраняются.

Замечание

При закрытии приложения событие `BeforeClose` не генерируется, и несохраненные изменения теряются.

Наборы данных могут находиться в различных *режимах*. Текущий режим набора данных определяется свойством `State` типа `TDataSetState`. Оно доступно для чтения при выполнении приложения и может быть использовано только для текущего режима. Для перевода набора данных в требуемый режим используются специальные методы. Они могут вызываться явно (указанием имени метода) или косвенно (путем управления соответствующими визуальными компонентами, например, навигатором `DBNavigator` или сеткой `DBGrid`).

Набор данных может находиться в одном из перечисленных ниже режимов.

- `dsInactive` (неактивен) — набор данных закрыт и доступ к его данным невозможен. В этот режим набор данных переходит после своего закрытия, когда свойству `Active` установлено значение `false`.
- `dsBrowse` (навигация по записям набора данных и просмотр данных) — в этот режим набор данных переходит так:
  - из режима `dsInactive` — при установке свойству `Active` значения `True`;
  - из режима `dsEdit` — при вызове метода `Post` или `Cancel`;
  - из режима `dsInsert` — при вызове метода `Post` или `Cancel`.
- `dsEdit` (редактирование текущей записи) — в этот режим набор данных переходит из режима `dsBrowse` при вызове метода `Edit`.
- `dsInsert` (вставка новой записи) — в этот режим набор данных переходит из режима `dsBrowse` при вызове методов `Insert`, `InsertRecord`, `Append` или `AppendRecord`.
- `dsSetKey` (поиск записи, удовлетворяющей заданному критерию) — в этот режим набор данных переходит из режима `dsBrowse` при вызове методов `SetKey`, `SetRangeXXX`, `FindKey`, `GotoKey`, `FindNearest` или `GotoNearest`. Возможен только

для компонента типа `TTable`, т. к. для компонента типа `TQuery` отбор записей осуществляется средствами языка SQL.

- `dsCalcFields` (расчет вычисляемых полей) — используется обработчик события `OnCalcFields`.
- `dsFilter` (фильтрация записей) — в этот режим набор данных автоматически переходит из режима `dsBrowse` каждый раз, когда выполняется обработчик события `OnFilterRecord`. В режиме блокируются все попытки изменения записей. После завершения работы обработчика события `OnFilterRecord` набор данных автоматически переводится в режим `dsBrowse`.
- `dsNewValue` (обращение к свойству `NewValue` компонента поля).
- `dsOldValue` (обращение к свойству `OldValue` компонента поля).
- `dsCurValue` (обращение к свойству `CurValue` компонента поля).
- `dsBlockRead` (запрет изменения элементов управления и генерации событий при вызове метода `Next`).
- `dsInternalCalc` (указание на необходимость вычислять значения полей, свойство `FieldKind` которых имеет значение `fkInternalCalc`).
- `dsOpening` (открытие набора данных).

Взаимосвязи между основными режимами наборов данных показаны на рис. 19.2, где приведены также некоторые методы и свойства, с помощью которых набор данных переходит из одного режима в другой.

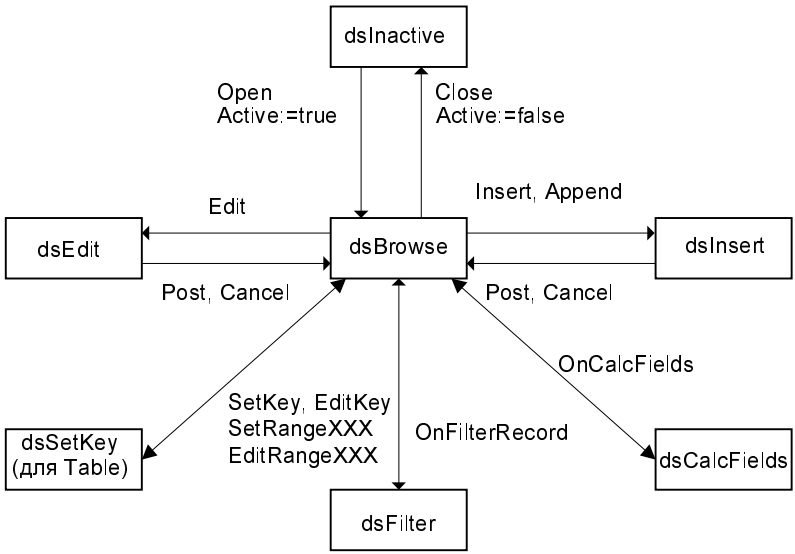


Рис. 19.2. Схема взаимосвязи режимов наборов данных

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 можно определить режим набора данных с помощью одноименных свойств `State` типа `TDataSetState` самого набора данных и связанного с ним источника данных `DataSource`. При изменении режима набора данных для источника данных `DataSource` генерируется событие `OnStateChange` типа `TNotifyEvent`.

Рассмотрим пример:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
begin
case DataSource1.State of
dsInactive: Label1.Caption := 'Набор данных закрыт';
dsBrowse:   Label1.Caption := 'Просмотр набора данных';
dsEdit:     Label1.Caption := 'Редактирование набора данных';
dsInsert:   Label1.Caption := 'Вставка записи в набор данных'
else        Label1.Caption := 'Режим набора данных не определен';
end;
end;
```

В приведенной процедуре определяется режим набора данных, связанного с источником данных `DataSource1`, и информация об этом режиме выводится в надписи `Label1`. При этом используется свойство `State` источника данных. Код, выполняющий анализ режима, помещен в обработчик события `OnStateChange` компонента `DataSource1`.

Особенности компонента *Table*

Компонент `Table` представляет собой набор данных, который в текущий момент времени может быть связан только с *одной* таблицей БД. Этот набор данных формируется на базе навигационного способа доступа к данным, поэтому компонент `Table` рекомендуется использовать для локальных БД, таких как dBase или Paradox. При работе с удаленными БД следует использовать компонент `Query`.

Связь между таблицей и компонентом `Table` устанавливается через его свойство `TableName` типа `TFileName`, которое задает имя таблицы (и имя файла с данными таблицы). При задании свойства `TableName` указываются имя файла и расширение имени файла.

На этапе разработки приложения имена всех таблиц доступны в раскрывающемся списке Инспектора объектов. В этот список попадают таблицы, файлы которых расположены в каталоге, указанном свойством `DatabaseName`.

Замечание

При смене имени таблицы на этапе проектирования приложения свойство `Active` набора данных автоматически устанавливается в значение `False`. При задании имени таблицы программным способом набор данных предварительно необходимо закрыть, установив его свойство `Active` в значение `False`. В противном случае генерируется исключение.

Приведем пример, иллюстрирующий, как задается имя таблицы БД:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if OpenDialog1.Execute then begin
 Table1.Active := False;
 Table1.TableName := OpenDialog1.FileName;
 Table1.Active := True;
 end;
end;
```

Здесь нажатие кнопки `Button1` приводит к появлению диалогового окна выбора имени файла. При выборе файла таблицы его имя устанавливается в качестве значения свойства `TableName`. Набор данных `Table1` предварительно закрывается и снова открывается уже после смены таблицы. Тип таблицы определяется автоматически по расширению имени файла. При наличии ошибок, например, связанных с нарушением структуры таблицы, выдается соответствующее сообщение, а набор данных остается закрытым.

Свойство `TableType` типа `TTtableType` определяет тип таблицы. Для локальных таблиц это свойство может принимать следующие значения:

- `ttASCII` — текстовый файл, содержащий данные в табличном виде (таблица ASCII);
- `ttDBase` — таблица dBase;
- `ttDefault` — тип таблицы автоматически определяется по расширению файла;
- `ttFoxPro` — таблица FoxPro;
- `ttParadox` — таблица Paradox.

Если свойство `TableType` имеет значение `ttDefault` (по умолчанию), то тип таблицы определяется по расширению файла:

- db или отсутствует — таблица Paradox;
- dbf — таблица dBase;
- txt — текстовый файл (таблица ASCII).

По умолчанию в состав набора данных `Table` попадают все записи связанной с ним таблицы. Для отбора записей, удовлетворяющих определенным условиям, используются фильтры.

Механизм BDE.NET автоматически поддерживает многопользовательский доступ к локальным таблицам, при этом по умолчанию все пользовательские приложения имеют равные права и могут редактировать содержащиеся в таблицах данные. Чтобы запретить пользователям изменять содержание записей, можно использовать свойство `ReadOnly` типа `Boolean`. По умолчанию оно имеет значение `False`, что предоставляет пользователю право изменения записей.

В наборе данных `Table` можно указать текущий индекс для выполнения операций:

- сортировки записей;
- поиска записей;
- установления связей между таблицами.

Текущий индекс устанавливается с помощью свойства `IndexName` или `IndexFieldNames` типа `String`. На этапе разработки приложения текущий индекс выбирается в списке индексов, заданных при создании таблицы. Все возможные значения свойств `IndexName` и `IndexFieldNames` содержатся в раскрывающихся списках, доступных через Инспектор объектов. Оба свойства во многом схожи, и их использование практически одинаково. Значением свойства `IndexName` является *имя индекса*, заданное при создании таблицы, а значением свойства `IndexFieldNames` является *имя поля*, для которого был создан индекс. Если индекс состоит из нескольких полей, то для свойства `IndexName` по-прежнему задается имя этого индекса, а для свойства `IndexFieldNames` через точку с запятой перечисляются имена полей, входящих в этот индекс.

Например, текущий индекс в программе можно задать так:

```
Table1.IndexName := 'indName';
Table2.IndexFieldNames := 'Name';
```

Здесь компоненты `Table1` и `Table2` связаны с одной таблицей, для поля `Name` которой определен индекс `indName`. Этот индекс устанавливается в качестве текущего для обоих наборов данных.

Для таблиц Paradox сделать текущим индексом ключ (главный индекс) можно только с помощью свойства `IndexFieldNames`, перечислив ключевые поля таблицы, т. к. ключ не имеет имени и поэтому недоступен через свойство `IndexName`.

Задать ключ в качестве текущего индекса можно так:

```
Table1.IndexFieldNames := 'Name;BirthDay';
```

Здесь для таблицы Paradox, с которой связан компонент `Table1`, определен ключ, в который входят поля `Name` и `BirthDay`. Этот ключ устанавливается в качестве текущего индекса таблицы.

Замечание

Свойства `IndexName` и `IndexFieldNames` взаимозависимы. При установке значения одного из них другое автоматически очищается.

Если текущий индекс, задаваемый как значение свойства `IndexName` или `IndexFieldNames`, для таблицы не существует, то возникает исключение.

При смене таблицы, с которой ассоциирован компонент `Table`, значения свойств `IndexName` и `IndexFieldNames` не изменяются автоматически, поэтому программист должен установить нужные значения самостоятельно.

Получить доступ к полям в составе текущего индекса можно с помощью свойств `IndexFieldCount` и `IndexFields`.

Особенности компонента *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.

Напомним, что 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. Для компонента `Query` можно указать в запросе SQL путь доступа к каждой таблице.

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. 19.3), вызываемый через Инспектор объектов двойным щелчком в области значения свойства `SQL`.

SQL-запрос также можно формировать и изменять динамически, внося изменения в его текст (значение свойства `SQL` компонента `Query`) 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.

Рассмотрим пример приложения — простейшего редактора, позволяющего подготавливать и выполнять SQL-запросы. На рис. 19.4 показана форма приложения при его выполнении. Кроме визуальных компонентов, форма содержит два компонента доступа к данным `Query1` и `DataSource1`, которые при выполнении приложения не видны.

Редактирование SQL-запроса осуществляется с помощью компонента `Memo1`. Набранный запрос выполняется при нажатии кнопки **Выполнить** (`Button1`), а результат выполнения отображается в компоненте `DBGrid1`.

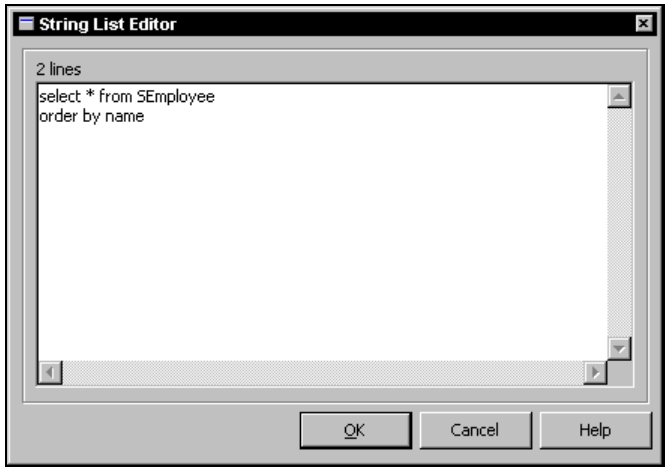


Рис. 19.3. Редактирование запроса SQL

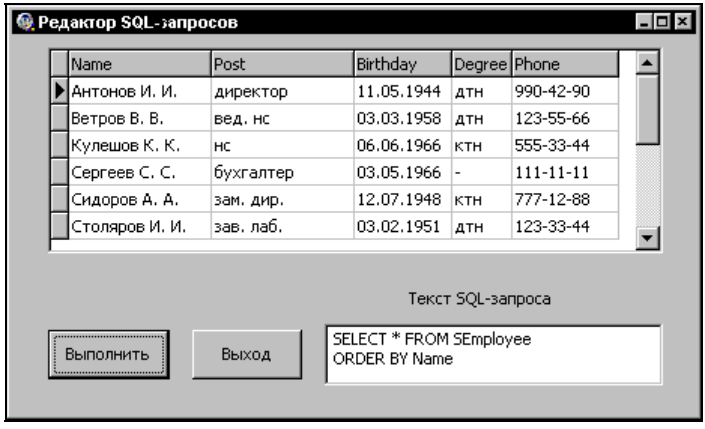


Рис. 19.4. Приложение-редактор SQL-запросов

При наличии в тексте SQL-запроса ошибки генерируется исключение и выдается сообщение об ошибке, а результат запроса оказывается не определен. При этом набор данных `Query1` автоматически закрывается.

Значения свойств `DataSet` источников данных `DataSource1` и `DataSource` компонента `DBGrid1`, с помощью которых организуется взаимодействие компонентов `Query1`, `DataSource1` и `DBGrid1`, устанавливаются при создании формы. В последующих примерах приложений значения этих свойств задаются через Инспектор объектов, поэтому инструкции, присваивающие свойствам необходимые значения, в модуле формы отсутствуют.

Приведем код модуля `Unit3` формы `Form3` приложения VCL.NET.

```
unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, Borland.Vcl.Grids, Borland.Vcl.DBGrids, Borland.Vcl.Db,
  Borland.Vcl.DBTables, System.ComponentModel, Borland.Vcl.StdCtrls;

type
  TForm3 = class(TForm)
    Button1: TButton;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    Label1: TLabel;
    Button2: TButton;
    Memo1: TMemo;
    Query1: TQuery;
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form3: TForm3;

implementation

{$R *.nfm}

procedure TForm3.Button1Click(Sender: TObject);
begin
Query1.DatabaseName:='DBDEMOS';
Query1.Close;
Query1.SQL.Assign(Memo1.Lines);
Query1.Open;
end;

procedure TForm3.Button2Click(Sender: TObject);
begin
Self.Close;
end;
```

```
procedure TForm3.FormCreate(Sender: TObject);
begin
// Эти действия можно выполнить
// с помощью Инспектора объектов
DataSource1.DataSet := Query1;
DBGrid1.DataSource := DataSource1;
end;

end.
```

Метод `Assign` выполняет присваивание одного объекта другому, при этом объекты должны иметь совместимые типы. Применительно к списку строк (класс `TStrings`), которому принадлежат свойства `SQL` компонента `Query1` и `Lines` компонента `Memo1`, подобное присваивание означает копирование информации из одного списка в другой с заменой содержимого последнего. Если размеры списков (число элементов) не совпадают, то после замены число элементов заменяемого списка становится равным числу элементов копируемого списка.

Компонент `Query` обеспечивает выполнение SQL-запроса и является набором данных, который формируется на основе этого запроса. Формирование набора данных выполняется при активизации компонента `Query` вызовом метода `Open` или установкой свойства `Active` в значение `True`. В ряде случаев при выполнении SQL-запроса не требуется возвращать набор данных, например, при удалении, вставке или изменении записей. В этом случае предпочтительнее выполнять запрос компонента `Query` не его открытием, а вызовом метода `ExecSQL`. При работе в сети вызов метода `ExecSQL` выполняет требуемую модификацию набора данных, не передавая в вызывающее приложение (компьютер) записи набора данных, что заметно снижает нагрузку на сеть.

Компонент `Query` может быть связан с таблицей БД или напрямую, или содержать *копии* отобранных записей таблицы, доступные для чтения. Вид взаимодействия определяется свойством `RequestLive` типа `Boolean`. По умолчанию свойство имеет значение `False`, и набор данных `Query` доступен только для чтения. Если пользователю или программисту требуется возможность редактирования записей, то 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 свойства `RequestLive`, можно воспользоваться свойством `CanModify` типа `Boolean`. Если оно имеет значение `True`, то набор данных является редактируемым, если `False` — то нередактируемым.

Чтобы получить в результате выполнения SQL-запроса редактируемый набор данных, кроме установки 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 `NameTableBD` является именем таблицы БД, которое для компонента `Table` задается в свойстве `TableName`.) Однако набор данных `Query`, в отличие от `Table`, не имеет системы индексов, поэтому к `Query` неприменимы методы, использующие индексирование, например, методы `FindFirst`, `FindLast`, `FindNext` и `FindPrior`.

Объекты-поля

Каждое поле набора данных представляет собой отдельный столбец, для работы с которым в приложениях VCL.NET служат объект `Field` типа `TField` и объекты производных от него типов, например, `TIntegerField`, `TFloatField` или `TStringField`. Для доступа к этим объектам и, соответственно, к полям записей у набора данных есть специальные методы и свойства, доступные при выполнении приложения.

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, 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. 19.1 приведены основные типы объектов `Field`.

Таблица 19.1. *Основные типы объектов* `Field`

Тип объекта	Вид поля
`TBLOBField`	Поле BLOB-значения (BLOB, Binary Large Object — большой двоичный объект)
`TMemoField`	Memo-поле (поле комментария)
`TGraphicField`	Графическое поле
`TDateTimeField`	Поле значения даты и времени
`TNumericField`	Поле числового значения
`TBCDField`	Поле BCD-значения

Таблица 19.1 *(окончание)*

Тип объекта	Вид поля
`TFloatField`	Поле вещественного значения
`TCurrencyField`	Поле значения денежной суммы
`TIntegerField`	Поле целочисленного значения
`TAutoincField`	Поле автоинкрементного значения
`TStringField`	Поле строкового значения

Задать состав полей набора данных можно двумя способами:

- по умолчанию (динамические поля);
- с помощью редактора полей (статические поля).

По умолчанию при каждом открытии набора данных на этапе проектирования 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.

Недостатки использования динамических полей состоят в том, 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 исключения.

Редактор полей

По умолчанию для каждого физического поля при открытии набора данных автоматически создается объект типа `TField`, а все поля в наборе данных являются динамическими и доступными. Для создания статических (устойчивых) полей используется специальный Редактор полей. В случае, если хотя бы одно поле набора данных является статическим, динамические поля больше создаваться не будут. Таким образом, в наборе данных будут доступны только статические поля, а все остальные считаются отсутствующими. Определить или отменить состав статических полей можно с помощью Редактора полей на этапе разработки приложения.

Замечание

Для компонента `Query` состав полей определяется также в тексте SQL-запроса, с помощью которого можно задать или изменить состав полей набора данных, несмотря на то, что эти поля являются динамическими.

Для запуска Редактора полей (рис. 19.5) следует сделать двойной щелчок на компоненте `Table` или `Query` или вызвать для этих компонентов контекстное меню правой кнопкой мыши и выбрать пункт **Fields Editor**. В заголовке Редактора полей выводится составное имя набора данных, например, `Form3.Query1`. Для перемещения по полям используются клавиши управления курсором или мышь. Большую часть этого редактора занимает список статических полей, при этом поля перечисляются в порядке их создания, который может отличаться от порядка полей в таблице БД.

Первоначально список статических полей пуст, указывая, что все поля набора данных являются динамическими.

С помощью Редактора полей разработчик может выполнить следующие операции:

- создать новое статическое поле;
- удалить статическое поле;
- изменить порядок следования статических полей.

Рис. 19.5. Редактор полей

Кроме того, для любого выбранного в редакторе статического поля с помощью Инспектора объектов можно задать или изменить свойства этого поля (объекта типа `TField`) и определить обработчики его событий. Подобные действия разрешается производить благодаря тому, что соответствующие статическим полям объекты типа `TField` доступны на этапе разработки приложения.

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 поле в списке и нажать клавишу <Delete>. После удаления статического поля оно становится недоступным для операций в программе, однако в случае необходимости его снова можно сделать статическим, добавив в список Редактора полей. При этом следует иметь в виду, что все свойства этого поля устанавливаются заново, а все сделанные ранее изменения теряются.

Замечание

Если удалены все статические поля, то все поля набора данных становятся динамическими и доступными при выполнении приложения.

Порядок следования полей определяется их местом в списке Редактора полей. По умолчанию порядок полей соответствует порядку физических полей в таблицах БД. Его можно изменить, перемещая поля в списке с помощью мыши или комбинаций клавиш <Ctrl>+<Page Up> и <Ctrl>+<Page Down>.

Замечание

Если для набора данных определены статические поля, то изменение значения свойства `TableName` этого набора данных может привести к ошибке, что обычно и происходит. Это связано с тем, что в новой таблице, связываемой с набором данных, могут отсутствовать физические поля, для которых были созданы статические поля. В таких случаях программист должен предусматривать соответствующие операции, например, формирование нового состава статических полей.

Есть три типа статических полей:

- поле данных, связанное с соответствующим физическим полем таблицы;
- вычисляемое поле, значение которого рассчитывается в обработчике события `OnCalcFields` во время выполнения приложения;
- поле выбора, значение которого можно выбирать из списка, формируемого на основе заданных критериев и правил.

Для создания нового статического поля любого типа нужно выбрать в контекстном меню Редактора полей пункт **New Field**, в результате чего открывается одноименное диалоговое окно (рис. 19.6).

Для задания общих свойств (параметров) нового поля используется группа элементов управления **Field properties** (Свойства поля). В поле ввода **Name** задается значение свойства `FieldName` (имя поля), а в поле ввода **Component** — значение

свойства `Name` (имя компонента поля — объекта типа `TField`). При программировании обычно используется имя поля. Значение в поле ввода **Component** формируется автоматически, и попытка изменить его не приводит к желаемому результату. В списке **Type** и поле ввода **Size** указываются тип данных и размер поля. Тип данных обязательно задается для всех полей, а необходимость задания размера зависит от типа данных. Например, для поля с типом данных `Integer` задание размера не имеет смысла, а для типа `String` размер поля ограничивает максимальную длину строки.

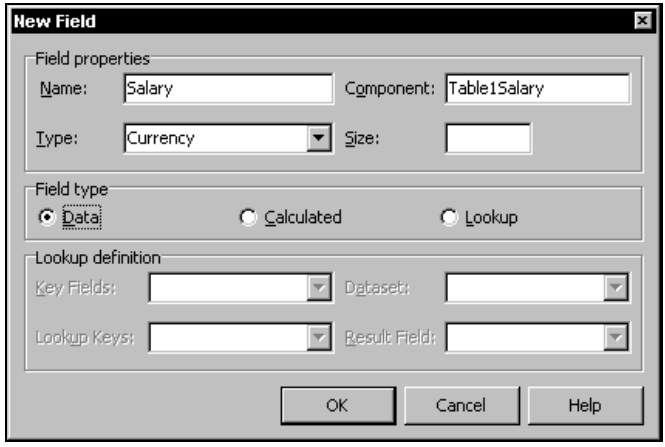


Рис. 19.6. Окно создания статического поля

Тип нового поля выбирается в группе переключателей **Field type** из следующих вариантов:

- **Data** (поле данных);
- **Calculated** (вычисляемое поле);
- **Lookup** (поле выбора).

В группе **Lookup definition** (Определение выбора) для поля выбора устанавливаются такие параметры, как набор данных и поля связи, а также поля для формирования списка выбора и результата.

Операции с полями

Через объект типа `TField` разработчик может:

- обратиться к полю и его значению;
- проверить тип и значение поля;
- отформатировать значение поля, отображаемое или редактируемое в визуальных компонентах.

При этом динамические и статические поля имеют одинаковые свойства, события и методы, с помощью которых можно управлять этими объектами при выполнении приложения. В связи с тем, что статические поля определяются на этапе разработки, многие их свойства доступны с помощью Инспектора объектов.

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 с помощью Инспектора объектов, когда соответствующее статическое поле выбрано в Редакторе полей.

В отличие от имени объекта `Field`, свойство `FieldName` типа `String` содержит имя *физического поля*, заданное при создании таблицы. Не следует путать свойства `Name` и `FieldName`, они обозначают разные объекты и в общем случае могут не совпадать.

Пример обращения к полю:

```
Table1.FieldByName('R_Salary').DisplayLabel := 'Оклад';
Table1R_Salary.DisplayLabel := 'Оклад';
```

Здесь для статического поля `R_Salary` приведены два возможных способа обращения: по имени поля в наборе данных и по имени объекта `Field` поля. Заметим, что свойство `DisplayLabel` определяет текст, отображаемый в качестве заголовка поля набора данных.

Для определения порядкового номера поля в наборе данных можно использовать свойство `FieldNo` типа `Integer`, например, так:

```
var x: integer;
...
x := Table1.FieldByName('Date').FieldNo;
```

Для *доступа* к значению поля служат свойства `Value` и `AsXXX`. Свойство `Value` типа `Variant` представляет собой фактические данные в объекте типа `TField`. При выполнении приложения это свойство используется для чтения и записи значений в поле. Если программист обращается к свойству `Value`, то он должен самостоятельно обеспечивать преобразование и согласование типов значений полей и читаемых или записываемых значений.

Поскольку при доступе к полю с помощью свойства `Value` программист должен обеспечивать преобразование и согласование типов значений, то часто более удобно использовать варианты свойства `AsXXX`, например: `AsVariant` типа `Variant`;

AsString типа String и др. При использовании любого из этих свойств выполняется *автоматическое преобразование типа* значения поля к типу, соответствующему названию свойства. При этом преобразование должно быть допустимо, в противном случае возникает ошибка компиляции по несоответствию типов.

Рассмотрим пример, в котором доступ к значению поля происходит с помощью свойств AsXXX.

```
procedure TForm3.Button1Click(Sender: TObject);
var s: string;
    x: integer;
begin
// Доступ к полю по его имени в наборе данных
s := Table1.FieldByName('Zip').AsString;
x := Table1.FieldByName('Zip').AsInteger;
Label1.Caption := s;
Label2.Caption := IntToStr(x);
// Доступ к полю как к отдельному компоненту
s := Table1Zip.AsString;
x := Table1Zip.AsInteger;
Label3.Caption := s;
Label4.Caption := IntToStr(x);
end;
```

Здесь чтение значения поля Zip осуществляется несколькими способами. Доступ к полю выполняется по имени поля и по имени объекта поля, а значение поля интерпретируется как строковое или как целочисленное.

Замечание

Чтобы записать значение в поле, оно должно допускать модификацию, а набор данных должен находиться в соответствующем режиме, например, редактирования или вставки.

При необходимости программист может запретить модификацию поля, а также скрыть его, используя свойства ReadOnly и Visible типа Boolean. Сама возможность модификации данных в отдельном поле определяется значением свойства CanModify типа Boolean. Напомним, что свойства ReadOnly и CanModify есть также у набора данных: они определяют возможность модификации набора данных (всех его полей) в целом.

Замечание

Даже если набор данных является модифицируемым и его свойство CanModify имеет значение True, для отдельных полей этого набора редактирование может быть запрещено, и любая попытка изменить значение такого поля вызовет исключение.

Если поле является невидимым (свойство Visible имеет значение False), но разрешено для редактирования (свойство ReadOnly имеет значение False), то можно изменить 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 `LoadFromFile`.

Операции с данными

Операции с данными рассматриваются на примере использования навигационного способа доступа к локальным базам данных с помощью механизма BDE. 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:

- сортировку записей;
- навигацию по набору данных;
- редактирование записей;
- вставку и удаление записей;
- фильтрацию записей.

Отметим, что аналогичные операции применимы к набору данных и при реляционном способе доступа, реализуемом с помощью SQL-запроса.

Сортировка записей

Порядок расположения записей в наборе данных может быть неопределенным. По умолчанию записи не отсортированы или сортируются, например, для таблиц Paradox по ключевым полям, а для таблиц dBase в порядке их поступления в файл таблицы.

С отсортированными записями набора данных работать более удобно. Сортировка заключается в упорядочивании записей по определенному полю в порядке возрастания или убывания содержащихся в нем значений. Сортировку можно выполнять и по нескольким полям. Например, при сортировке по двум полям сначала записи упорядочиваются по значениям первого поля, а затем группы записей с одинаковым значением первого поля сортируются по второму полю.

Сортировка наборов данных `Table` и `Query` выполняется различными способами. Здесь мы рассмотрим сортировку набора данных `Table` для компонента `Query` 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 признак `ixDescending` индекса включен, то сортировка выполняется в порядке убывания значений.

Напомним, что задать индекс (текущий индекс), по которому выполняется сортировка записей, можно с помощью свойств `IndexName` или `IndexFieldNames`. Эти свойства являются взаимоисключающими, и установка значения одного из них приводит к автоматической очистке значения другого. В качестве значения свойства `IndexName` указывается имя индекса, установленное при его создании. При использовании свойства `IndexFieldNames` указываются имена полей, образующих соответствующий индекс.

В связи с тем, что главный индекс (ключ) таблиц Paradox не имеет имени, выполнить сортировку по этому индексу можно только с помощью свойства `IndexFieldNames`.

Приведем пример сортировки с указанием имен индексов и индексных полей:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
case RadioGroup1.ItemIndex of
     0: Table1.IndexFieldNames := 'Товар';
     1: Table1.IndexName := 'indОплата';
     2: Table1.IndexFieldNames := 'Валюта';
     end;
end;
```

В качестве набора данных используется компонент `Table1`, а сортировка выполняется тремя способами: по имени индексного поля `Товар` (для связанной с набором данных таблицы поле `Товар` определено в качестве главного индекса), по индексу `indОплата`, созданному для поля `Оплата` и по имени индексного поля `Валюта` (индекс `indВалюта`).

Возможный вид формы приложения, обеспечивающего указанные варианты сортировки, приведен на рис. 19.7. В качестве набора данных используется компонент `Table1`. Пользователь может управлять сортировкой его записей с помощью группы переключателей, посредством которой определяется вид сортировки. Сортировка выполняется после нажатия кнопки **Задать сортировку** (`Button2`).

При необходимости изменить порядок сортировки по некоторому полю, это удобно сделать либо при настройке свойств таблицы с помощью программы

Database Desktop, либо при создании индексов программно путем задания значения `ixDescending` для параметра `Options`, например так:

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

Рис. 19.7. Вид формы для сортировки набора данных

Управление направлением сортировки осуществляется с помощью параметра `ixDescending` текущего индекса. Для определения номера текущего индекса в списке `IndexDefs` используется метод `IndexOf`.

Навигация по набору данных

Навигация по набору данных заключается в управлении указателем текущей записи (курсором). Этот указатель определяет запись, с которой будут выполняться такие операции, как редактирование или удаление.

Перед перемещением указателя текущей записи набор данных автоматически переводится в режим просмотра. Если текущая запись находилась в режимах ре-

дактирования или вставки, то перед перемещением указателя сделанные в записи изменения вступают в силу, для чего набор данных автоматически вызывает метод `CheckBrowseMode`.

Для перемещения указателя текущей записи в наборе данных используются следующие методы:

- процедура `First` — установка на первую запись;
- процедура `Next` — установка на следующую запись (при вызове метода для последней записи указатель не перемещается);
- процедура `Last` — установка на последнюю запись;
- процедура `Prior` — установка на предыдущую запись (при вызове метода для первой записи указатель не перемещается);
- функция `MoveBy (Distance: Integer): Integer` — перемещение на число записей, определяемое параметром `Distance`. Если его значение больше нуля, то перемещение осуществляется вперед, если меньше нуля — то назад. При нулевом значении параметра указатель не перемещается. Если заданное параметром `Distance` число записей выходит за начало или конец набора данных, то указатель устанавливается на первую или на последнюю запись. В качестве результата возвращается число записей, на которое переместился указатель.

При перемещении указателя текущей записи учитываются ограничения и фильтр, определенные для набора данных. Таким образом, перемещение выполняется по записям набора данных, которые он содержит в текущий момент времени. Число записей определяется свойством `RecordCount`.

Замечание

При изменении порядка сортировки набора данных расположение его записей может измениться, что чаще всего и происходит, но указатель по-прежнему указывает на первоначальную запись, даже если она находится на другом месте и имеет новое значение свойства `RecNo`.

Рассмотрим следующий пример:

```
procedure TForm1.Button3Click(Sender: TObject);
var s: real;
    n: integer;
begin
s := 0;
// Установка текущего указателя на первую запись
Table1.First;
for n := 1 to Table1.RecordCount do begin
    s := s + Table1.FieldByName('Стоимость').AsFloat;
    // Перемещение текущего указателя на следующую запись
    Table1.Next;
    end;
```

```
Label2.Caption := FloatToStr(s);
end;
```

В приведенной процедуре перебираются все записи набора данных Table1, при этом в переменной s накапливается сумма значений, содержащихся в поле Стоимость. 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.

Аналогичным образом можно перебрать все записи набора данных, начиная с последней. Естественно, при этом нужно вызывать методы Last и Prior.

Для *контроля положения* указателя текущей записи можно использовать свойство RecNo, которое содержит номер записи, считая от начала набора данных (для локальных таблиц dBase и Paradox).

Для таблиц Paradox свойство RecNo можно использовать еще и для перехода к записи с известным номером: такой переход выполняется установкой свойства RecNo в значение, равное номеру нужной записи. Например:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Table1.RecNo := StrToInt(Edit1.Text);
end;
```

При нажатии кнопки Button1 указатель текущей записи набора данных Table1 устанавливается на запись, номер которой содержит редактор Edit1.

Для определения *начала* и *конца* набора данных при перемещении указателя текущей записи можно использовать свойства BOF и EOF типа Boolean соответственно. Эти свойства доступны для чтения при выполнении приложения. Свойство BOF показывает, 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.

При работе с таблицами одновременно нескольких приложений, когда постоянно добавляются или удаляются записи, значения свойств Bof и Eof соответствуют действительному состоянию набора данных в определенные моменты времени. Так,

свойству Eof устанавливается значение True сразу после выполнения метода Last. Если после этого другим приложением в конец набора данных добавлена новая запись, то значение свойства Eof становится неправильным.

При перемещении по записям набора данных связанные с ним визуальные компоненты отображают изменения данных, причем смена отображения может происходить достаточно быстро, вызывая неприятное мелькание на экране. Чтобы избежать этого эффекта, можно программно до начала цикла перебора записей временно отключить набор данных от всех связанных с ним визуальных компонентов, а по окончании цикла снова подключить. Для этого предназначены методы DisableControls и EnableControls.

Пользователь имеет возможность *перемещаться* по набору данных с помощью элементов управления, в качестве которых часто используются компоненты DBGrid и DBNavigator. Управление этими элементами с помощью мыши или клавиатуры приводит к автоматическому вызову соответствующих методов, перемещающих указатель текущей записи в заданное место. Например, после нажатия кнопок **First record**, **Prior record**, **Next record** или **Last record** компонента DBNavigator1 косвенно вызываются методы First, Prior, Next или Last, перемещающие указатель текущей записи соответственно на первую, предыдущую, следующую или последнюю записи набора данных, с которым связан (через источник данных DataSource) компонент DBNavigator1.

Другим вариантом является размещение на форме элементов управления, например, кнопки Button, предназначенных для навигации по набору данных. Кроме того, часто на форме также размещаются элементы (обычно кнопки) для управления такими операциями, как редактирование, вставка записей, фильтрация и сортировка набора данных.

Фильтрация записей

Фильтрация — это задание ограничений для записей, отбираемых в набор данных. Напомним, что набор данных представляет собой записи, выбранные из одной или нескольких таблиц. Состав записей в наборе данных в определенный момент времени зависит от установленных ограничений, в том числе от фильтров. 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, поскольку просто ограничивает число записей, видимых в наборе. Кроме того, на выполнение фильтрации затрачиваются ресурсы. В приложениях VCL.NET можно осуществлять фильтрации записей:

- по выражению;
- по диапазону.

Рассмотрим *фильтрацию по выражению* как более универсальную, при использовании которой набор данных ограничивается записями, удовлетворяющими выражению фильтра, задающему условия отбора записей. Достоинством фильтрации по выражению является то, что она применима к любым полям, в том числе к неиндексированным. В связи с тем, что в процессе отбора просматриваются все записи таблицы, фильтрация по выражению эффективна при небольшом количестве записей.

Для задания выражения фильтра используется свойство `Filter` типа `String`. Фильтр представляет собой конструкцию, в состав которой могут входить следующие элементы:

- имена полей таблиц;
- литералы;
- операции сравнения;
- арифметические операции;
- логические операции;
- круглые и квадратные скобки.

Если имя поля содержит пробелы, то его заключают в квадратные скобки, в противном случае квадратные скобки необязательны.

Литерал представляет собой значение, заданное явно, например, число, строка или символ. Отметим, что имена переменных в выражении фильтра использовать нельзя. Если в фильтр требуется включить значение переменной или свойства какого-либо компонента, то это значение должно быть преобразовано в строковый тип.

Операции сравнения представляют собой обычные для языка Pascal отношения `<`, `>`, `=`, `<=`, `>=` и `<>`.

Арифметическими являются операции `+`, `-`, `*` и `/` (сложения, вычитания, умножения и деления соответственно).

В качестве логических операций можно использовать `AND`, `OR` и `NOT` (логическое умножение, сложение и отрицание соответственно).

Круглые скобки применяются для изменения порядка выполнения арифметических и логических операций.

В качестве примеров задания условий фильтрации приведем следующие выражения:

```
Salary <= 2000
Post = 'Лаборант' OR Post = 'Инженер'
```

Первое выражение обеспечивает отбор всех записей, для которых значение поля оклада (`Salary`) не превышает `2000`. Второе выражение обеспечивает отбор записей, поле должности (`Post`) которых содержит значение `Лаборант` или `Инженер`.

Если выражение фильтра не позволяет сформировать сложный критерий фильтрации, то в дополнение к нему можно использовать обработчик события `OnFilterRecord`.

Для *активизации* и *деактивизации* фильтра применяется свойство `Filter` типа `Boolean`. По умолчанию это свойство имеет значение `False`, и фильтрация выключена. При установке свойства `Filtered` в значение `True` фильтрация включается, и в набор данных отбираются записи, которые удовлетворяют фильтру, записанному в свойстве `Filter`. Если выражение фильтра не задано (по умолчанию), то в набор данных попадают все записи.

Замечание

Активизация фильтра и выполнение фильтрации возможны также на этапе разработки приложения.

Если выражение фильтра содержит ошибки, то при попытке выполнить его генерируется исключение. Если фильтр неактивен (свойство `Filtered` имеет значение `False`), то выражение фильтра не анализируется на корректность.

Параметры фильтрации задаются с помощью свойства `FilterOptions` типа `TFilterOptions`. Это свойство принадлежит к множественному типу и может принимать комбинации двух значений:

- `foCaseInsensitive` — регистр букв не учитывается, т. е. при задании фильтра `Post = 'Водитель'` слова `Водитель`, `ВОДИТЕЛЬ` или `водитель` будут восприняты как одинаковые. Значение `foCaseInsensitive` нужно отключать, если требуется различать слова, написанные в различных регистрах;
- `foNoPartialCompare` — выполняется проверка на полное соответствие содержимого поля и значения, заданного для поиска. Обычно применяется для строк символов. Если известны только первые символы (или символ) строки, то нужно указать их в выражении фильтра, заменив остальные символы на звездочки (`*`) и выключив значение `foNoPartialCompare`. Например, при выключенном значении `foNoPartialCompare` для фильтра `Post = '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 `[]`.

Рассмотрим в качестве примера обработчики событий формы, используемой для фильтрации записей набора данных по выражению. Вид формы приведен на рис. 19.8.

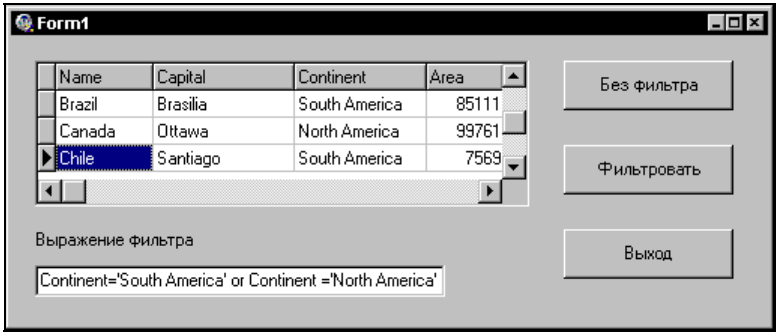


Рис. 19.8. Фильтрация по выражению

Управление фильтрацией набора данных выполняется с помощью двух кнопок и поля редактирования. При нажатии кнопки **Фильтровать** (`btnFilter`) фильтр активизируется путем присваивания значения `True` свойству `Filtered` набора данных. Редактор `edtFilter` предназначен для задания выражения фильтра.

При активизации фильтра происходит отбор записей, которые удовлетворяют заданному в выражении условию. При нажатии кнопки **Без фильтра** (`btnNoFilter`) фильтр отключается, при этом показываются все записи.

Выражения для фильтра может содержать сложные условия формирования фильтра, в том числе с помощью логических операций `OR` (см. рис. 19.8) и `NOT`.

Далее приведены три обработчика событий модуля формы `Form1` приложения.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
Table1.FilterOptions := [foCaseInsensitive];
Table1.Filtered := False;
end;
procedure TForm1.btnFilterClick(Sender: TObject);
begin
Table1.Filtered := True;
Table1.Filter := edtFilter.Text;
end;
procedure TForm1.btnAllRecordClick(Sender: TObject);
begin
Table1.Filtered := False;
end;
```

Включение и выключение фильтра осуществляется через свойство `Filtered`. Фильтрация выполняется без учета регистра букв.

В приведенном примере пользователь должен самостоятельно набирать выражение фильтра. Это предоставляет пользователю достаточно широкие возможности управления фильтрацией, но требует от него знания правил построения выражений.

Например, чтобы задать фильтрацию по значению поля типа даты, можно использовать следующий обработчик события:

```
procedure TForm3.Button1Click(Sender: TObject);
begin
table1.filter:= 'BIRTH_DATE<'+QuotedStr('10.10.1955');
table1.filtered:=true;
end;
```

Здесь при фильтрации обеспечивается отбор записей, в которых значение поля `BIRTH_DATE` (дата рождения) содержит дату более раннюю, чем `10.10.1955`. То есть отбираются данные о сотрудниках, родившихся до указанной даты. Для ввода значений даты удобно использовать компоненты типа `TDateTimePicker`, с помощью которых пользователь может выбирать дату.

Поскольку в выражении фильтра нельзя использовать имена переменных и свойств компонентов, то при его формировании набранные пользователем значения должны преобразовываться в строковый тип. В приведенном примере это выполняется с помощью функции `QuotedStr`.

Часто удобно предоставить пользователю список готовых выражений (шаблонов) для выбора. При этом пользователь получает также возможность редактировать выбранное выражение и корректировать весь список. Такой режим реализуется, например, с помощью компонентов `ComboBox` и `Memo`.

Если набор условий фильтрации ограничен и не изменяется, то пользователь может управлять отбором записей с помощью таких компонентов, как флажки (`CheckBox`) и переключатели (`RadioButton`).

Поиск записей

Поиск записи, удовлетворяющей определенным условиям, означает переход на эту запись. Поиск во многом похож на фильтрацию, т. к. в процессе поиска также выполняется проверка полей записей по некоторым условиям. Отличие заключается в том, что в результате поиска количество записей в наборе данных не изменяется.

При организации поиска записей важно наличие индекса для полей, по которым ведется поиск. Индексирование значительно повышает скорость обработки данных, кроме того, ряд методов может работать только с индексированными полями. Таким образом, можно выделить методы поиска записей по полям и по индексным полям.

Далее мы рассмотрим средства, с помощью которых можно выполнить поиск записей в наборах данных `Table` и `Query`. Отметим, что к средствам поиска можно отнести также методы `FindFirst`, `FindLast`, `FindNext` и `FindPrior`, осуществляющие переход на записи, удовлетворяющие условиям фильтра.

Для поиска записей по полям служат методы `Locate` и `Lookup`, причем поля могут быть неиндексированными.

Функция `Locate (const KeyFields: String; const KeyValues: Variant; Options: TLocateOptions): Boolean` ищет запись с заданными значениями полей. Если удовлетворяющие условиям поиска записи существуют, то указатель текущей записи устанавливается на первую из них. Если запись найдена, функция возвращает значение `True`, в противном случае — значение `False`. Список полей, по которым ведется поиск, задается в параметре `KeyFields`, поля разделяются точкой с запятой. Параметр `KeyValues` типа `Variant` указывает значения полей для поиска. Если поиск ведется по одному полю, то параметр содержит одно значение, соответствующее типу поля, заданного для поиска.

Параметр `Options` позволяет задать значения, которые обычно используются при поиске строк. Этот параметр принадлежит к множественному типу `TLocateOptions` и принимает комбинации следующих значений:

- `loCaseInsensitive` (регистр букв не учитывается);
- `loPartialKey` (допускается частичное совпадение значений).

Отметим, что тип `TLocateOptions` по сути похож на тип `TFilterOptions`, определяющий параметры фильтрации по выражению, но значения `loPartialKey` и `foNoPartialCompare` имеют противоположное действие: первое из них допускает, а второе запрещает частичное совпадение значений.

Замечание

При наличии у параметра `Options` значения `loPartialKey` к нему автоматически добавляется значение `loCaseInsensitive`.

Пример поиска по одному полю:

```
procedure TForm3.Button1Click(Sender: TObject);
begin
Table1.Locate('Capital', 'Lima', []);
end;
```

Поиск выполняется по полю `Capital` и ищется первая запись, для которой значением этого поля является строка `'Lima'`. Все параметры поиска отключены. Возвращаемый методом `Locate` результат не анализируется.

При поиске *по нескольким полям* в методе `Locate` параметр `KeyValues` является массивом значений типа `Variant`, в котором содержится несколько элементов. Для приведения к типу вариантного массива используется функция `VarArrayOf`.

Значения разделяются запятыми и заключаются в квадратные скобки, порядок значений должен соответствовать порядку полей параметра KeyFields. Пример:

```
Table1.Locate('Capital;Continent;', VarArrayOf(['M', 'America']),
[loCseInsensitive, loPartialKey]);
```

Поиск выполняется по полям Capital и Continent, ищется первая запись, для которой значение поля столицы начинается с букв m или M, а значение поля континента содержит строку America. Регистр букв значения не имеет. Результат поиска не анализируется.

Замечание

Значение loPartialKey параметра Options должно обеспечивать возможность частичного совпадения строковых значений в ключевых полях, по которым осуществляется поиск. В действительности при поиске по нескольким полям поиск записи происходит только при строгом совпадении значений для всех ключевых полей.

При поиске только по одному полю (как в предпоследнем примере) значение loPartialKey параметра Options обеспечивает возможность частичного совпадения строковых значений.

Обычно при разработке приложений пользователю предоставляется возможность влиять на процесс поиска с помощью элементов управления, расположенных на форме. При этом действия пользователя по управлению поиском в наборе данных мало 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 различие между двумя методами заключается в том, что метод Lookup осуществляет поиск на *точное* соответствие значений для поиска и значений в полях записей с учетом регистра букв.

Параметры KeyFields и KeyValues имеют такое же назначение, как и в методе Locate, и используются аналогичным образом.

В параметре ResultFields через точку с запятой перечисляются названия полей, значения которых будут получены в случае успешного поиска. Эти значения считываются из первой найденной записи, удовлетворяющей условиям поиска. Порядок перечисления полей в ResultFields может отличаться от порядка полей в наборе данных. Например, если набор данных имеет поля Code, Name, Salary и Note, то в ResultFields можно задать Salary и Name.

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.

Модификация набора данных

Модификация (изменение) набора данных представляет собой редактирование, добавление и удаление его записей. Модифицируемость набора данных зависит от различных условий. Разработчик может разрешить или запретить изменение набора данных с помощью соответствующих свойств.

Управлять возможностью изменения набора данных Table можно посредством свойства ReadOnly типа Boolean, при установке которого в значение True изменение записей запрещается. По умолчанию свойство ReadOnly имеет значение False, и набор данных можно модифицировать.

Замечание

Значение свойства ReadOnly можно изменять только у закрытого набора данных.

В отличие от многих элементов управления, например, редакторов Edit и Memo, данный запрет на редактирование относится как к визуальному (пользователем), так и к программному изменению записей набора данных.

Для проверки, можно ли изменять набор данных, предназначено свойство CanModify типа Boolean, действующее при выполнении приложения и доступное только для чтения. Если это свойство имеет значение True, то набор данных изменять можно, а если False, то изменения в наборе данных запрещены, и любая попытка сделать это визуально или программно вызовет исключение.

Замечание

Можно запретить редактирование отдельных полей набора данных даже в том случае, если он является модифицируемым.

Для программного изменения набора данных вызываются соответствующие методы, например, метод Edit редактирования текущей записи или метод Append вставки новой записи.

Пользователь редактирует набор данных с помощью визуальных компонентов, например, редактора DBEdit или сетки DBGrid, управляя ими с помощью мыши и клавиатуры. Набор данных может автоматически переводиться в режимы редактирования или вставки, для этого свойство AutoEdit источника данных DateSource для визуальных компонентов должно быть установлено в значение True (по умолчанию). Если 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 элементы управления, такие как, например, кнопка Button или флажок CheckBox.

При модификации набора данных для связанного с ним источника данных DateSource генерируется событие OnUpdateData.

После модификации набора данных возможна ситуация, когда изменения сделаны, но не отображены визуальными компонентами, связанными с этим набором. В таких случаях нужно вызывать метод Refresh, который повторно считывает набор данных и тем самым гарантирует, что визуальные компоненты будут отображать текущие, а не устаревшие данные. При работе в многопользовательской системе рекомендуется периодически вызывать метод Refresh, чтобы своевременно учитывать изменения, сделанные другими пользователями.

В однопользовательском приложении обновление набора данных применяется в случае, если с одной таблицей связано несколько наборов данных. Например, с таблицей клиентов может быть связан набор данных, с помощью которого выполняется редактирование списка клиентов, а также набор данных, предназначенный для выбора клиента при вводе информации в расходную накладную. Компоненты Table обоих наборов могут находиться на 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.

Приведем пример кода, в котором проводится обновление набора данных:

```
Table1.Edit;
Table1.FieldByName('Capital').AsString:=Edit1.Text;
Table1.Post;
Table1.Refresh;
Table2.Refresh;
```

В этом примере при программном изменении набора данных Table1, находящегося на форме Form1, обновляется он сам, а также набор данных Table2, расположенный тоже на форме Form1. Оба набора связаны с одной и той же физической таблицей.

Редактирование записей

Редактирование записей заключается в изменении значений их полей. Отредактировать можно только текущую запись, поэтому перед редактированием обычно выполняются операции поиска и перемещения на требуемую запись. Если указатель текущей записи установлен на нужную запись, и набор данных находится в режиме просмотра, для редактирования записи следует:

1. Перевести набор данных в режим редактирования.
2. Изменить значения полей записи.

3. Подтвердить сделанные изменения или отказаться от них, в результате чего набор данных снова перейдет в режим просмотра.

Набор данных переводится в режим редактирования вызовом метода `Edit`, при этом возможны такие ситуации:

- если набор данных немодифицируемый, то возбуждается исключение;
- если набор данных уже находился в режиме редактирования или вставки, то никакие действия не выполняются;
- если набор данных пуст, то он переходит в режим вставки.

Если набор данных является модифицируемым и исключение не возбуждается, то при выполнении метода `Edit` производятся следующие действия:

1. Для набора данных вызывается обработчик события `BeforeEdit` типа `TDataSetNotifyEvent`.
2. Из набора данных заново считывается текущая запись и блокируется так, чтобы другие пользователи могли ее читать, но не могли изменять или блокировать. Если операция блокирования завершается неудачно, то генерируется исключение, а выполнение метода `Edit` прекращается.
3. Если в записи есть вычисляемые поля, то они пересчитываются.
4. Набор данных переходит в режим редактирования.
5. Для связанного с набором данных источника данных `DataSource` вызывается обработчик события `OnDataChange`.
6. Для набора данных вызывается обработчик события `AfterEdit` типа `TDataSetNotifyEvent`.

Указанные действия осуществляются только для модифицируемого набора данных, поэтому перед вызовом метода `Edit` следует выполнять проверку на возможность редактирования записи, например, путем анализа свойства `CanModify`:

```
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` надо сделать двойной щелчок на нужном поле или нажать алфавитно-цифровую клавишу, когда курсор находится в этом поле, а для компонента `DBNavigator` требуется нажать кнопку **Edit Record**. Таким образом, при управлении визуальными компонентами метод `Edit` вызывается пользователем косвенно. В случае, когда набор данных является немодифицируемым, блокировка перехода в режим его редактирования выполняется автоматически и не приводит к ошибке.

Для компонентов управления, *не связанных* с набором данных, например, кнопок `Button` или флажков `CheckBox` программист должен самостоятельно кодировать

действия по предотвращению попыток редактирования немодифицируемого набора данных.

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 можно предложить подтвердить свои действия. Для отмены процесса редактирования в обработчике события `BeforeEdit` можно сгенерировать "тихое" исключение.

При переходе в режим редактирования обычно удобнее всего использовать обработчик события `BeforeEdit`, т. к. оно генерируется при переводе набора данных в режим редактирования любым способом. Например:

```
procedure TForm1.Table1BeforeEdit(DataSet: TDataSet);
begin
if MessageDlg('Выполнить редактирование?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Abort;
end;
```

После перевода набора данных в режим редактирования можно с помощью инструкций присваивания изменять значения полей текущей записи. При этом нужно учитывать тип поля, выполняя при необходимости операции приведения типов. Например:

```
Table1.FieldByName('City').AsString := Edit1.Text;
Table1.FieldByName('Area').AsInteger := StrToInt(Edit2.Text);
```

Перед выполнением приведенных инструкций набор данных `Table1` должен находиться в режиме редактирования или вставки. Если редактор `Edit2` содержит данные в формате, не соответствующем целому числу, то генерируется исключение.

Для проверки, вносились ли изменения в запись, можно проанализировать свойство `Modified` типа `Boolean`. Если свойство имеет значение `True`, то было изменено значение как минимум одного поля текущей записи.

После ввода информации сделанные изменения должны быть или подтверждены, или отменены.

Метод `Post` *записывает модифицированную запись* в таблицу БД, снимает блокировку записи и переводит набор данных в режим просмотра. Если набор данных не находился в режиме редактирования, то вызов метода `Post` приведет к генерации исключения. Перед его выполнением автоматически вызывается обработчик события `BeforePost` типа `TDataSetNotifyEvent`, а сразу после выполнения — обработчик события `AfterPost` типа `TDataSetNotifyEvent`. Используя событие `BeforePost`,

можно проверить сделанные изменения и при необходимости отменить их, например, прервав выполнение метода с помощью вызова "тихого" исключения.

Приведем процедуру, в которой осуществляется редактирование записей:

```
procedure TForm3.Button1Click(Sender: TObject);
begin
// Изменение площади в текущей записи
  Table1.Edit;
  Table1.FieldByName('Area').AsInteger := StrToInt(Edit1.Text);
  Table1.Post;
// Переход к следующей записи
  Table1.Next;
end;
```

Здесь поле `Area` текущей записи набора данных получает новое значение из свойства `Text` компонента `Edit1`.

Метод `Post` вызывается автоматически при переходе к другой записи с помощью методов `First`, `Last`, `Next` и `Prior`, если набор данных находится в режиме редактирования, и изменения в записях не подтверждены. Поэтому в приведенном примере метод `Post` можно было не вызывать, т. к. сразу после него вызывается метод `Next`. Однако при использовании методов `FindFirst`, `FindLast`, `FindNext` и `FindPrior` неподтвержденные изменения в записях будут потеряны.

Пользователь подтверждает сделанные в записях изменения, управляя соответствующими компонентами, явно или неявно вызывающими метод `Post`. Конкретные действия пользователя зависят от используемых компонентов. Например, при работе с компонентом `DBGrid` изменения подтверждаются (фиксируются) при переходе к другой записи или нажатии клавиши <Enter>, а в компоненте `DBNavigator` можно нажать кнопку **Post Edit** (Подтвердить изменения).

Независимо от способа вызова, метод `Post` может завершиться неудачно, например, если не заданы значения полей, которые не могут быть пустыми, или значение выходит за установленные для него допустимые пределы. В этом случае набор данных обычно возвращается в состояние, которое было до перехода в режим редактирования. При *ошибке выполнения* метода `Post` генерируется событие `OnPostError` типа `TDataSetErrorEvent`. Кодируя обработчик этого события, можно попытаться исправить ошибку.

Метод `Cancel` *отменяет изменения*, выполненные в текущей записи, и возвращает набор данных в режим просмотра. При выполнении метода `Cancel` вызываются обработчики событий `BeforeCancel` и `AfterCancel` типа `TDataSetNotifyEvent`.

Пользователь может отменить сделанные в записях изменения, используя элементы управления компонентов. Например, при работе с сеткой `DBGrid` изменения отменяются нажатием клавиши <Esc>, а в компоненте `DBNavigator` — нажатием кнопки **Cancel Edit**.

При *редактировании* текущей записи последовательность инструкций присваивания можно заменить вызовом метода `SetFields`.

Метод имеет объявление `procedure SetFields(const Values: array of Object);` и устанавливает все или часть значений полей текущей записи. Параметр `Values` содержит массив значений, которые присваиваются этим полям, при этом порядок значений соответствует порядку полей в наборе данных. Кроме того, должны соответствовать типы полей и типы присваиваемых им значений. Если значений в массиве меньше, чем полей в наборе данных, то эти значения присваиваются первым полям, а оставшиеся поля не изменяются. Если в качестве значения элемента массива задать `nil`, то соответствующее поле примет значение 0 или будет пустым в случае символьной строки. Если число значений в массиве превышает число полей, то при выполнении метода `SetFields` генерируется исключение.

Замечание

Если для набора данных использовался Редактор полей, то при выполнении метода `SetFields` учитывается порядок полей, установленный с помощью этого Редактора. В противном случае принимается порядок полей, определенный при создании таблицы БД.

Метод `SetFields` удобно использовать для изменения значений нескольких полей. После его выполнения набор данных автоматически возвращается в режим просмотра. Пример использования метода:

```
procedure TForm3.Button1Click(Sender: TObject);
begin
Table1.Edit;
Table1.SetFields(['Guyana', 'Georgetown', nil, 344567, 164700000]);
Table1.Post;
end;
```

Здесь в первое, второе, четвертое и пятое поля текущей записи набора данных `Table1` заносятся страна, столица, площадь территории и численность населения соответственно. Третье поле этой записи принимает значение пустой строки.

Вставка и добавление записей

Вставлять и добавлять записи можно только к модифицируемому набору данных, в противном случае будет сгенерировано исключение.

Для вставки или добавления записи нужно выполнить следующие действия:

1. Перевести набор данных в режим вставки.
2. Задать значения полей новой записи и выполнить вставку или добавление.
3. Подтвердить сделанные изменения или отказаться от них, после чего набор данных снова переходит в режим просмотра.

Для вставки записей используются методы `Insert`, `InsertRecord`, для добавления записей — методы `Append` и `AppendRecord`.

Метод `Insert` переводит набор данных в режим вставки и вставляет новую пустую запись в позицию непосредственно перед указателем текущей записи. При необходимости перед вызовом метода `Insert` нужно выполнить перемещение текущего указателя в требуемую позицию набора данных.

После перевода набора данных в режим вставки дальнейшие действия по заданию (изменению) значений полей, подтверждению или отмене сделанных изменений не отличаются от аналогичных действий при редактировании записи. При этом для задания или изменения значений полей используются инструкции присваивания и метод `SetFields`, а для подтверждения или отмены изменений — методы `Post` и `Cancel`. Некоторые поля новой записи могут остаться пустыми, если до подтверждения им не были присвоены значения.

Замечание

При переходе в режим вставки к набору данных добавляется пустая запись, значения полей которой не заданы. Если запись добавляется к подчиненному набору данных, связанному с главным набором связью "главный-подчиненный", то индексные поля автоматически получают корректные значения, и программист может не заботиться об их заполнении.

Рассмотрим следующий пример:

```
procedure TForm3.Button1Click(Sender: TObject);
Table1.Insert;
Table1.FieldByName('Country').AsString := Edit1.Text;
Table1.Post;
end;
```

Здесь в новой записи задается значение поля страны (`Country`), 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.

Метод `InsertRecord` объединяет функциональность методов `Insert` и `SetFields`, выполняя те же действия, что и их последовательный вызов. Процедура `InsertRecord (const Values: array of const)` вставляет в позицию перед указателем текущей записи новую запись, задавая значения всех или части ее полей. Например:

```
procedure TForm1.btnInsertClick(Sender: TObject);
begin
Table1.InsertRecord([Edit1.Text, 'Georgetown', nil, 344567, 164700000]);
end;
```

Здесь выполняется вставка новой записи в позицию перед указателем текущей записи набора данных `Table1`, затем в первое, второе, четвертое и пятое поля но-

вой записи заносятся страна (из компонента Edit1.Text), столица, площадь территории и численность населения соответственно. Третье поле этой записи принимает значение пустой строки.

Методы Append и AppendRecord отличаются от методов Insert и InsertRecord тем, что добавляют запись в конец набора данных, а не в позицию перед указателем текущей записи.

Пользователь управляет набором данных, в том числе вставкой и добавлением записи, с помощью элементов управления формы. Для компонента DBGrid новая запись добавляется к набору данных при нажатии клавиши <Insert> или при переходе на последнюю запись. Если на форме находится компонент DBNavigator, то новая запись добавляется при нажатии кнопки **Insert Record**.

При вставке или добавлении новой записи любым методом возникают события BeforeInsert, AfterInsert и OnNewRecord типа TDataSetNotifyEvent.

Событие AfterInsert возникает сразу после вставки или добавления новой записи, его удобно использовать, например, для заполнения полей новой записи.

Событие BeforeInsert возникает перед вставкой или добавлением новой записи, его можно использовать для контроля данных перед заполнением полей новой записи или для модификации полей текущей записи.

В обработчике события OnNewRecord можно выполнить действия, связанные с проверкой набранных пользователем данных или с заполнением (инициализацией) части полей новой записи.

Приведем пример использования обработчика события AfterInsert:

```
procedure TForm3.Table1AfterInsert(DataSet: TDataSet);
begin
Table1.Edit;
Table1.FieldByName('Name').AsString := Edit1.Text;
Table1.FieldByName('Area').AsInteger := StrToInt(Edit2.Text);
Table1.Post;
end;
```

Здесь сразу после вставки или добавления новой записи производится заполнение двух полей новой записи строковым (компонент Edit1.Text) и целочисленным (компонент Edit2.Text) значениями.

При утверждении или отмене изменений, связанных с добавлением новой записи, также генерируются события BeforePost и AfterPost или BeforeCancel и AfterCancel.

Удаление записей

Удаление текущей записи выполняет метод Delete, который работает только с модифицируемым набором данных. В случае успешного удаления записи текущей становится следующая запись, если же удалялась последняя запись, то кур-

сор перемещается на предыдущую запись, которая после удаления становится последней. В отличие от некоторых СУБД, в Delphi .NET удаляемая запись действительно удаляется из набора данных.

Обычно метод `Delete` вызывается для удаления просматриваемой записи, однако с его помощью можно удалить и редактируемую запись. Если набор данных находится в режиме вставки или поиска, то вызов метода `Delete` аналогичен вызову метода `Cancel`, отменяя соответственно вставку или поиск записи.

Замечание

Если набор данных пуст, то вызов метода `Delete` порождает исключение.

При удалении записи генерируются события `BeforeDelete` и `AfterDelete` типа `TDataSetNotifyEvent`. Используя обработчик события `BeforeDelete`, можно отменить операцию удаления, если не соблюдаются определенные условия. Например:

```
procedure TForm3.Table1BeforeDelete(DataSet: TDataSet);
begin
if MessageDlg('Удалить запись?',
       mtConfirmation, [mbYes, mbNo], 0) = mrNo then Abort;
end;
```

Здесь в обработчике события `BeforeDelete` перед удалением записи запрашивается подтверждение пользователя. При не подтверждении запроса вызывается процедура `Abort`, которая генерирует "тихое" исключение.

Если выполнение метода `Delete` приводит к ошибке, то возбуждается исключение, 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 перебору удаляемых записей не нужно вызывать методы, перемещающие указатель текущей записи. После удаления текущей записи указатель автоматически перемещается на соседнюю (обычно следующую) запись. Так можно удалить все записи набора данных:

```
procedure TForm1.btnDeleteAllClick(Sender: TObject);
var n: longint;
begin
Table1.Last;
for n := Table1.RecordCount downto 1 do Table1.Delete;
end;
```

В примере перебор записей выполняется с конца набора данных. После удаления текущей записи указатель снова оказывается на последней записи.

Для набора данных `Table` удалить все записи можно также с помощью метода `EmptyTable`, который вызывается в режиме исключительного доступа к таблице БД.

Перед удалением записи часто предварительно выполняется поиск записей, удовлетворяющих заданным условиям. Для отбора группы удаляемых записей используется фильтрация. Метод `Delete` позволяет удалить записи, видимые в наборе данных. Поэтому с помощью фильтрации можно временно оставить в наборе данных записи, которые подлежат удалению, а после удаления фильтрацию отключить.

Пример формы приложения

Обычно визуальные компоненты, предназначенные для редактирования, добавления и удаления записей, группируются на форме и работают взаимосвязанно. Вместе или рядом с этими компонентами часто располагают элементы управления сортировкой, фильтрацией и поиском. Тем самым для пользователя обеспечивается удобство выполнения различных операций с данными.

Для примера рассмотрим форму (рис. 19.9), с помощью которой можно изменять записи таблицы, содержащей сведения о сотрудниках. Данные о сотрудниках включают фамилию (поле `Name`), должность (поле `Post`), дату рождения (поле `Birthday`), ученую степень (поле `Degree`) и телефон (поле `Pfone`). Поле фамилии и поле должности обязательно должны быть заполненными.

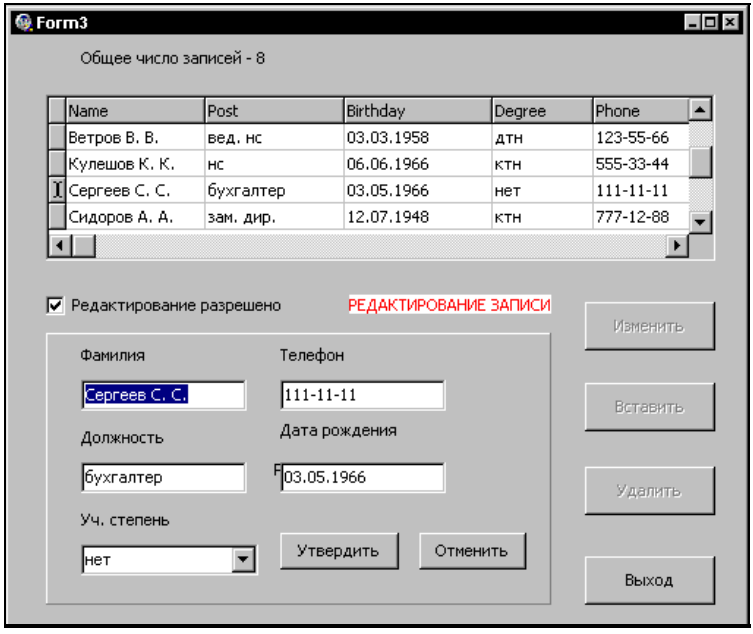


Рис. 19.9. Форма приложения для работы с данными о сотрудниках

Ниже приводится код модуля `Unit3FApp` формы `Form3` приложения.

```
unit Unit3FApp;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, Borland.Vcl.Db, Borland.Vcl.DBTables, Borland.Vcl.StdCtrls,
Borland.Vcl.DBCtrls, Borland.Vcl.Mask, Borland.Vcl.Grids, Borland.Vcl.DBGrids,
  System.ComponentModel, Borland.Vcl.ExtCtrls;

type
  TForm3 = class(TForm)
    Panel1: TPanel;
    DBGrid1: TDBGrid;
    DBEdit1Name: TDBEdit;
    DBEdit2Post: TDBEdit;
    DBEdit1Phone: TDBEdit;
    DBEdit1Birthday: TDBEdit;
    Button1Edit: TButton;
    Button2Insert: TButton;
    Button3Delete: TButton;
    Button4ChangeOK: TButton;
    Button5ChangeCancel: TButton;
    CheckBox1CanEdit: TCheckBox;
    Label1Regime: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label1RecordCount: TLabel;
    DBComboBox1Degree: TDBComboBox;
    Button6: TButton;
    Table1: TTable;
    DataSource1: TDataSource;
    procedure Button6Click(Sender: TObject);
    procedure Button5ChangeCancelClick(Sender: TObject);
    procedure Button4ChangeOKClick(Sender: TObject);
    procedure Table1BeforePost(DataSet: TDataSet);
    procedure Button3DeleteClick(Sender: TObject);
    procedure Button2InsertClick(Sender: TObject);
    procedure Table1BeforeInsert(DataSet: TDataSet);
    procedure Button1EditClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
```

```
    procedure CheckBox1CanEditClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SBrowse(Sender: TObject);
    procedure SChange(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form3: TForm3;

implementation

{$R *.nfm}

procedure TForm3.Button1EditClick(Sender: TObject);
begin
// Переход в режим редактирования
  Table1.Edit;
  Label1Regime.Font.Color := clRed;
  Label1Regime.Caption := 'РЕДАКТИРОВАНИЕ ЗАПИСИ';
  SChange(Sender);
  if DBEdit1Name.CanFocus then DBEdit1Name.SetFocus;
end;

procedure TForm3.Button2InsertClick(Sender: TObject);
begin
// Переход в режим вставки
         Table1.Insert;
         Label1Regime.Font.Color := clGreen;
         Label1Regime.Caption  := 'ВСТАВКА ЗАПИСИ';
         SChange(Sender);
         if DBEdit1Name.CanFocus then DBEdit1Name.SetFocus;
end;

procedure TForm3.Button3DeleteClick(Sender: TObject);
begin
// Подтверждение перехода в режим просмотра удаляемой записи
         if MessageDlg('Удалить запись?',
              mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Exit;
    Label1Regime.Font.Color := clRed;
         Label1Regime.Caption := 'УДАЛЕНИЕ ЗАПИСИ';
         SChange(Sender);
```

```
         if Button5ChangeCancel.CanFocus then
    Button5ChangeCancel.SetFocus;
end;

procedure TForm3.Button4ChangeOKClick(Sender: TObject);
begin
// Утверждение изменений в текущей записи (редактируемой или новой)
// или удаления текущей записи (просматриваемой)
        if Table1.State in [dsEdit, dsInsert]
           then Table1.Post
           else if Label1Regime.Caption = 'УДАЛЕНИЕ ЗАПИСИ' then
Table1.Delete;
        SBrowse(Sender);
end;

procedure TForm3.Button5ChangeCancelClick(Sender: TObject);
begin
// Если набор данных был в режиме просмотра (при удалении записи),
// то никаких действий метод Cancel не выполняет
         Table1.Cancel;
         SBrowse(Sender);
end;

procedure TForm3.Button6Click(Sender: TObject);
begin
  Close;
end;

procedure TForm3.CheckBox1CanEditClick(Sender: TObject);
var  bm:  TBookmark;
begin
         // Запоминание положения текущей записи
         bm := Table1.GetBookmark;
        // Отключение отображения изменений данных
        // в визуальных компонентах
         Table1.DisableControls;
         if not CheckBox1CanEdit.Checked then begin
             Table1.Active := false;
             Table1.ReadOnly := true;
             Table1.Active := true;
             // Блокирование элементов перехода
             // в режимы изменения записей
             Button1Edit.Enabled := false;
             Button2Insert.Enabled := false;
             Button3Delete.Enabled := false;
     end
```

```
        else   begin
          Table1.Active := false;
          Table1.ReadOnly := false;
          Table1.Active := true;
          // Разблокирование элементов перехода
          // в режимы изменения записей
          Button1Edit.Enabled := true;
          Button2Insert.Enabled := true;
          // Запрет удаления записей для пустого набора данных
          if Table1.RecordCount > 0
          then Button3Delete.Enabled := true
        else Button3Delete.Enabled := false;
          end;
          // Возврат к текущей записи
          if Table1.BookmarkValid(bm) then Table1.GotoBookmark(bm);
          if Table1.BookmarkValid(bm) then Table1.FreeBookmark(bm);
          // Включение отображения изменений данных
          // в визуальных компонентах
          Table1.EnableControls;

end;

procedure TForm3.FormCreate(Sender: TObject);
begin
 Table1.Active := true;
 // Исходное состояние элементов управления
 SBrowse(Sender);
 // Изменение записей запрещено
 CheckBox1CanEdit.Checked := false;
 // Запрет автоматического перехода в режим редактирования
 DataSource1.AutoEdit := false;
 // Формирование списка ученых степеней
 DBComboBox1Degree.Items.Clear;
 DBComboBox1Degree.Items.Add('нет');
 DBComboBox1Degree.Items.Add('ктн');
 DBComboBox1Degree.Items.Add('дтн');

end;

procedure TForm3.FormDestroy(Sender: TObject);
begin
  Table1.Active := false;
end;

// Режим просмотра
procedure TForm3.SBrowse(Sender: TObject);
```

```
begin
   // Процедура настройки режима просмотра записей
    Label1RecordCount.Caption := 'Общее число записей — ' +
IntToStr(Table1.RecordCount);
   // Вызов процедуры обработки события щелчка флажка CheckBox1CanEdit
   CheckBox1CanEditClick(Sender);
   Button4ChangeOK.Enabled := false;
   Button5ChangeCancel.Enabled := false;
   Label1Regime.Font.Color := clBlack;
   Label1Regime.Caption := 'ПРОСМОТР ЗАПИСИ';
end;

// Режимы изменения
procedure TForm3.SChange(Sender: TObject);
begin
   // Процедура настройки режима изменения записей
   Button1Edit.Enabled := false;
   Button2Insert.Enabled := false;
   Button3Delete.Enabled := false;
   Button4ChangeOK.Enabled := true;
   Button5ChangeCancel.Enabled := true;
end;

procedure TForm3.Table1BeforeEdit(DataSet: TDataSet);
begin
   // Запрос на подтверждение перехода в режим редактирования
   if MessageDlg('Будете редактировать?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Abort;
end;

procedure TForm3.Table1BeforeInsert(DataSet: TDataSet);
begin
   // Подтверждение перехода в режим вставки
   if MessageDlg('Добавить запись?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Abort;
end;

procedure TForm3.Table1BeforePost(DataSet: TDataSet);
begin
   // Проверка, задана ли фамилия
   if DBEdit1Name.Text = '' then begin
       Beep();
       MessageDlg('Фамилия не задана!', mtError, [mbOK], 0);
       if DBEdit1Name.CanFocus then DBEdit1Name.SetFocus;
       Abort;
     end;
```

```
   // Проверка, задана ли должность
   if DBEdit2Post.Text = '' then begin
       Beep();
       MessageDlg('Должность не задана!', mtError, [mbOK], 0);
       if DBEdit2Post.CanFocus then DBEdit2Post.SetFocus;
       Abort;
     end;
end;

end.
```

В качестве набора данных используется компонент `Table1`, записи которого отображает сетка `DBGrid1`. Значения полей текущей записи отображаются в компонентах `DBEdit1Name`, `DBEdit2Post`, `DBEdit1Phone`, `DBEdit1Birthday` и `DBComboBox1Degree`. Установка свойств, определяющих взаимные отношения компонентов, связанных с набором данных, выполнена при разработке приложения с помощью Инспектора объектов.

Пользователь не может переводить набор данных в режимы изменения с помощью визуальных компонентов, т. к. свойству `AutoEdit` источника данных `DataSource1` установлено значение `False`.

Для перевода набора данных в режимы редактирования и вставки служат кнопки **Изменить** (`Button1Edit`) и **Вставить** (`Button2Insert`) соответственно. Переход в эти режимы осуществляется вызовом методов `Edit` и `Insert`, после чего название режима отображается в надписи `Label1Regime`, блокируются кнопки, связанные с переходом в режимы изменения, разблокируются кнопки `Button4ChangeOK` и `Button5ChangeCancel`, позволяющие принять или отменить изменения, сделанные при редактировании или вставке записи. Для подтверждения или отмены изменений вызываются методы `Post` или `Cancel`, после чего кнопки подтверждения снова блокируются, а кнопки перехода в режимы изменения разблокируются. Изменение и задание значений полей выполняются с помощью компонентов `DBEdit1Name`, `DBEdit2Post`, `DBEdit1Phone`, `DBEdit1Birthday` и `DBComboBox1Degree`.

При переходе в режимы, связанные с изменением записей, пользователю предлагается подтвердить свои действия, что программируется в обработчиках событий `OnBeforeEdit` и `OnBeforeInsert`.

Программирование удаления записей отличается от предыдущих действий по изменению набора данных, т. к. режим удаления у набора данных отсутствует. При нажатии кнопки **Удалить** (`Button3Delete`) набор данных остается в режиме просмотра текущей записи. При этом `Label1Regime` указывает на переход в режим удаления, и если пользователь подтвердит удаление, нажав кнопку `Button4ChangeOK`, то для удаления текущей записи вызывается метод `Delete`. Запрос на подтверждение перехода в режим удаления кодируется в обработчике события нажатия кнопки `Button3DeleteClick`.

Флажок `CheckBox1CanEdit` с заголовком **Редактирование разрешено** включает и отключает возможность изменения в наборе данных. Управление этой возможностью осуществляется через свойство `ReadOnly` набора данных. Свойство переключается только при закрытом наборе данных. После нового открытия набора данных указатель текущей записи устанавливается на первую запись, поэтому перед переключением свойства `ReadOnly` положение указателя текущей записи запоминается с помощью закладки, а после переключения это положение восстанавливается. При разрешении изменений в наборе данных (в обработчике события смены состояния флажка `CheckBox1CanEdit`) проверяется число его записей, и если набор данных пуст, то кнопка `Button3Delete` блокируется.

Связывание таблиц

В общем случае приложение может иметь доступ к нескольким связанным таблицам. Как отмечалось, связь между таблицами устанавливается через поля связи, которые обязательно должны быть индексированными. При задании связи между двумя таблицами имеет место отношение подчиненности: 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" (Master-Detail). После установления связи между таблицами при перемещении в главной таблице текущего указателя на какую-либо запись в подчиненной таблице автоматически становятся доступными записи, у которых значение поля связи равно значению поля связи текущей записи главной таблицы.

Для организации связи между таблицами в подчиненной таблице (компонент `Table` в случае механизма BDE или его аналоги для других механизмов доступа) 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.

Во время работы со связанными таблицами нужно учитывать некоторые 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 ссылочной целостности (по изменению полей связи и каскадному удалению записей) могут быть наложены на таблицы при их создании, например, в среде программы Database Desktop, или устанавливаться программно.

Рассмотрим следующий пример. Предположим, что требуется установить связь между тремя таблицами. Две из них являются подчиненными (таблицы `PSalary` — оклад по должности и `DSarary` — надбавка за степень) и одна — главной (`SEmployee` — данные о сотрудниках) (рис. 19.10).

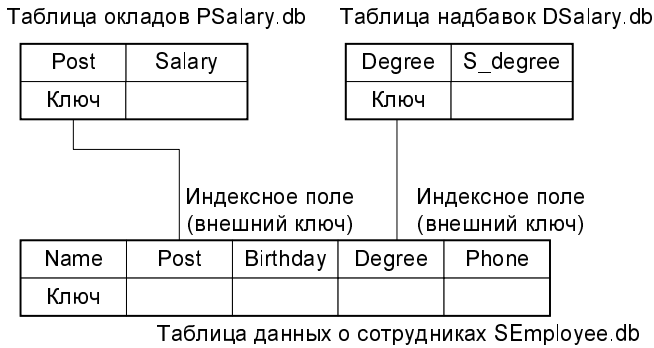


Рис. 19.10. Связи между таблицами

Для каждой из трех таблиц в приложении зададим по два компонента `Table` и `DataSource`. А именно `Table1` и `DataSource1` — для таблицы данных о сотрудниках, `Table2` и `DataSource2` — для таблицы окладов, `Table3` и `DataSource3` — для таблицы надбавок.

Для компонентов `Table1`—`Table3` в качестве значения свойства `DatabaseName` установим `DBDEMOS`, а свойству `TableName` зададим значения `SEmployee`, `PSalary` и `DSarary` соответственно.

Для компонентов `DataSource1`—`DataSource3` свойству `DataSet` выберем значения `Table1`—`Table3` соответственно.

Для каждого из компонентов подчиненных таблиц (`Table2` и `Table3`) свойству `MasterSource` путем выбора из списка зададим одинаковое значение `DataSource1`. Затем щелчком мыши в строке свойства `MasterFields` откроем диалоговое окно **Field Link Designer** (Конструктор полей связи) (рис. 19.11). Здесь в списке **Available Indexes** (Доступные индексы) выбирается индекс подчиненной таблицы, после чего составляющие его поля отображаются в списке **Detail Fields** (Детальные поля). В этом списке необходимо выбрать поле подчиненной таблицы, а в списке **Master Fields** (Главное поле) — поле главной таблицы (рис. 19.11). После нажатия кнопки **Add** выбранные поля связываются между собой, что ото-

бражается в списке **Joined Fields** (Связанные поля), например, так: `P_Post -> Post`. При этом оба поля пропадают из своих списков. Заполнение свойств `IndexName` и `MasterFields` происходит после закрытия окна при нажатии кнопки **OK**.

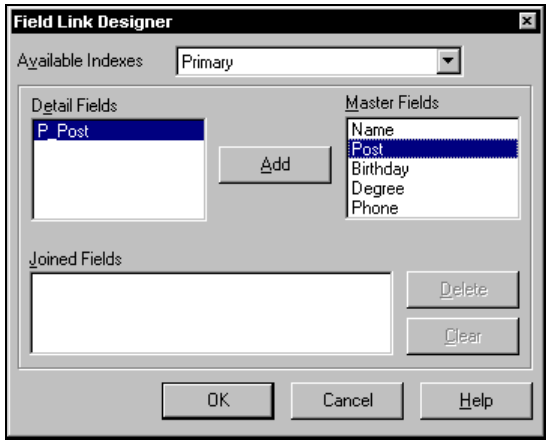


Рис. 19.11. Диалоговое окно Конструктора полей связи

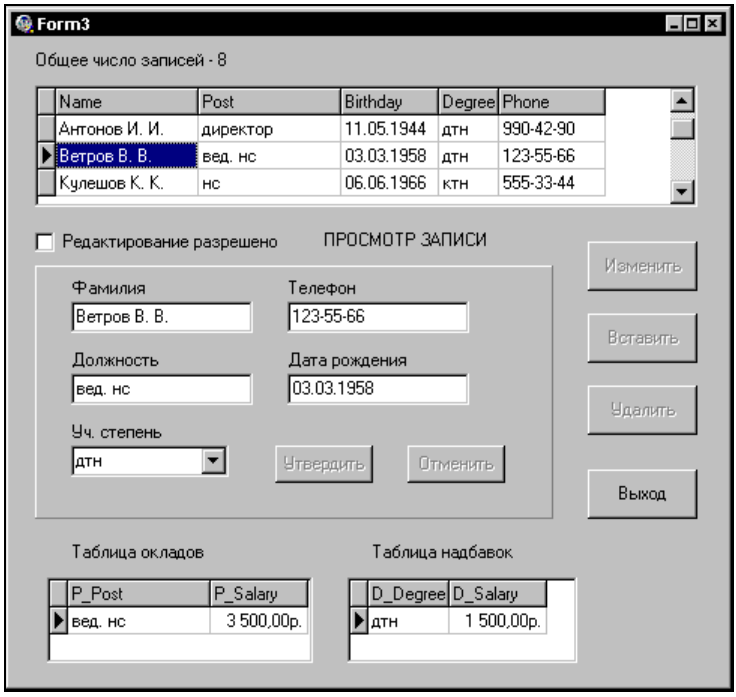


Рис. 19.12. Вид приложения с несколькими таблицами при выполнении

Установление связей между главной и подчиненными таблицами приводит к тому, что при выполнении приложения выбор некоторой записи в главной таблице будет приводить к выбору соответствующих записей в каждой из подчиненных таблиц (рис. 19.12).

В частности при выборе в главной таблице (рис. 19.12) записи со значением `вед. нс` поля `Post` в подчиненной таблице автоматически отображается строка с тем же значением в поле связи `P_Post`.

В рассмотренном нами примере использовалась связь типа "многие-к-одному". Чаще применяется связь типа "один-ко-многим". Однако в этом случае описанным способом можно установить связь только между двумя таблицами (одной главной таблицей и одной подчиненной таблицей), в то время как при данном типе связи требуется наличие нескольких главных и одной подчиненной таблиц.

ГЛАВА 20

Технология dgGo.NET

Общая характеристика

Технология dgGo.NET в среде Delphi .NET 2006 представляет собой новое название технологии Microsoft ActiveX Data Objects (ADO), введенное с той целью, чтобы отличить технологии ADO и ADO.NET. Она представляет собой универсальный механизм доступа к различным источникам данных из приложений баз данных. Основу технологии dgGo.NET (ADO) составляет использование набора интерфейсов общей модели объектов COM, описанных в спецификации OLE DB. Достоинством этой технологии является то, что базовый набор интерфейсов OLE DB имеется в каждой современной операционной системе Microsoft. Отсюда следует простота обеспечения доступа приложения к данным. Применяя технологию dgGo.NET (ADO), рис. 20.1, приложение БД может использовать данные из электронных таблиц, таблиц локальных и серверных баз данных, XML-файлов и т. д.

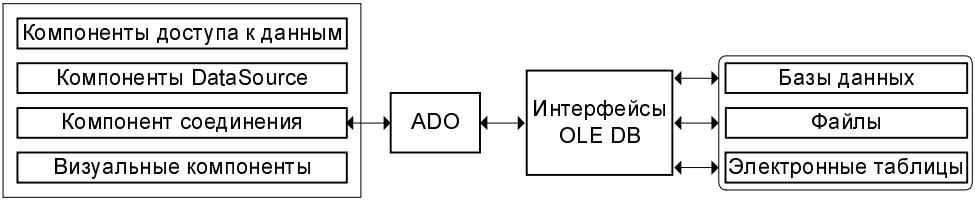


Рис. 20.1. Схема доступа к данным по технологии dgGo.NET (ADO)

В соответствии с терминологией ADO, любой источник данных (базу данных, файл, электронную таблицу) называют *хранилищем данных*. Приложение взаимодействует с хранилищем данных с помощью *провайдера*. Для каждого типа хранилища данных используется свой провайдер ADO. Провайдер обеспечивает обращение к данным хранилища с запросами, интерпретацию возвращаемой

служебной информации и результатов выполнения запросов для передачи их приложению.

Все объекты и интерфейсы ADO являются объектами и интерфейсами COM. Согласно спецификации OLE DB, в состав COM входит следующий набор объектов:

- `Command` (команда) — служит для обработки команд (обычно SQL-запросов);
- `Data Source` (источник данных) — используется для связи с провайдером данных;
- `Enumerator` (перечислитель) — служит для перечисления провайдеров ADO;
- `Error` (ошибка) — содержит информацию об исключениях;
- `Rowset` (набор рядов) — строки данных, являющиеся результатом выполнения команды;
- `Session` (сессия) — совокупность объектов, обращающихся к одному хранилищу данных;
- `Transaction` (транзакция) — управление транзакциями в OLE DB.

Мы привели состав объектов COM, чтобы получить общую картину о технологии ADO. В Delphi и Delphi .NET 2006 некоторые из этих объектов получили реализацию, и при необходимости мы будем рассматривать более подробно отдельные понятия.

В системе программирования Delphi .NET 2006 компоненты, используемые для создания приложений VCL.NET по технологии dgGo.NET (ADO), расположены на странице **dgGo** Палитры инструментов (рис. 20.2).

Рис. 20.2. Страница **dgGo**

Охарактеризуем кратко назначение этих компонентов (напомним, что префикс `T` в названии на странице **dgGo** указывает на тип компонента):

- `ADOConnection` (ADO-соединение) — используется для установки соединения с ADO-источником данных и обеспечивает поддержку транзакций;
- `ADOCommand` (ADO-команды) — используется для выполнения SQL-команд доступа к ADO-источнику данных без возвращения результирующего набора данных;

- `ADODataSet` (набор данных ADO) — обеспечивает доступ к одной (или более) таблице ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом `ADOTable` через компонент `DataSource` аналогично тому, как используется компонент `DataSet`. Может использоваться в компонентах `ADOTable`, `ADOQuery`, `ADOStoredProc`;
- `ADOTable` (таблица ADO) — обеспечивает доступ к одной таблице ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом `ADOTable` через компонент `DataSource`;
- `ADOQuery` (запрос ADO) — позволяет выполнять SQL-команды для получения информации из ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом `ADOTable` через компонент `DataSource`;
- `ADOStoredProc` (хранимая процедура ADO) — позволяет приложениям получать доступ к хранимым процедурам, используя интерфейс ADO;
- `RDSConnection` (RDS-соединение) — служит для управления передачей объекта `Recordset` от одного процесса (компьютера) к другому. Компонент используется для создания серверных приложений.

Компонент `ADOConnection` может использоваться как посредник между данными и другими компонентами ADO, что позволяет более гибко управлять соединением.

Возможен вариант использования других компонентов ADO путем соединения с источником данных напрямую, для чего компоненты ADO имеют свойство `ConnectionString`, с помощью которого могут создавать свой собственный канал доступа к данным.

Установление соединения

При использовании компонентов доступа к данным по технологии dgGo.NET (ADO) (`ADOCommand`, `ADODataSet`, `ADOTable`, `ADOQuery` и `ADOStoredProc`) установление соединения с хранилищем данных можно выполнить двумя путями: с помощью свойства `ConnectionString` или с помощью компонента `ADOConnection`. Имя последнего задается в свойстве `Connection` компонентов доступа к данным. При этом во втором случае для компонента `ADOConnection` предварительно также с помощью его свойства `ConnectionString` нужно установить соединение с хранилищем данных.

Рассмотрим технологию установления соединения с хранилищем данных с помощью свойства `ConnectionString`. Это свойство представляет собой строку с параметрами соединения, отделяемыми друг от друга точкой с запятой. Предварительно соответствующий компонент доступа к данным (например, `ADODataSet`) или компонент соединения (`ADOConnection`) должен быть помещен на форму при-

ложения. Настройка параметров соединения осуществляется в диалоге (рис. 20.3), открываемом двойным щелчком в строке `ConnectionString` свойства соответствующего компонента доступа к данным в окне Инспектора объектов.

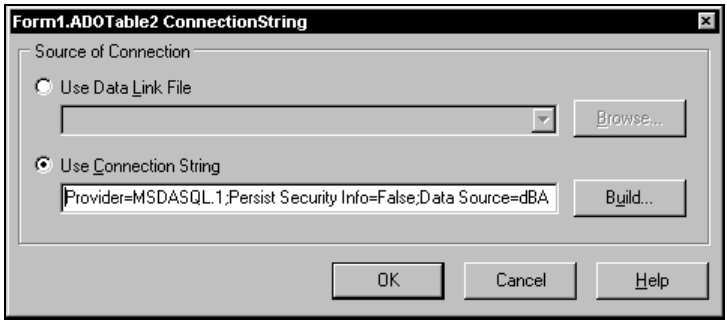


Рис. 20.3. Первое окно настройки строки соединения

При установке переключателя **Use Data Link File** можно выбрать из списка или найти (после нажатия кнопки **Browse**) *файл связи с данными* (расширение udl). По умолчанию он расположен в папке c:\Program Files\Common Files\System\Ole DB\Data Links. По существу файлы связи с данными играют ту же роль, что и псевдонимы при использовании BDE. Они позволяют разработчику не связывать откомпилированные приложения с точным расположением хранилища данных. При перемещении хранилища данных в другое место достаточно исправить содержимое файла связи с данными.

При установке переключателя **Use Connection String** выполняются действия по *созданию строки соединения*. Рассмотрим их более подробно. Для продолжения выбранного варианта диалога нужно нажать кнопку **Build**. В результате открывается диалоговое окно **Data Link Properties**, содержащее 4 вкладки. С помощью вкладки **Provider** (рис. 20.4) осуществляется выбор провайдера с учетом характера решаемой задачи. По умолчанию предлагается вариант **Microsoft OLE DB Provider for ODBC Drivers**.

Отметим, что после установки Microsoft ActiveX Data Objects в операционной системе доступны стандартные провайдеры ADO, обеспечивающие следующее:

- **Microsoft Jet OLE DB Provider** — соединение с данными СУБД Microsoft Access;
- **Microsoft OLE DB Provider for Microsoft Indexing Service** — доступ для чтения к ресурсам Microsoft Indexing Service;
- **Microsoft OLE DB Provider for Microsoft Active Directory Service** — доступ к ресурсам службы каталогов Active Directory;
- **Microsoft OLE DB Provider for Internet Publishing** — доступ к ресурсам Microsoft FrontPage и Microsoft Internet Information Server;

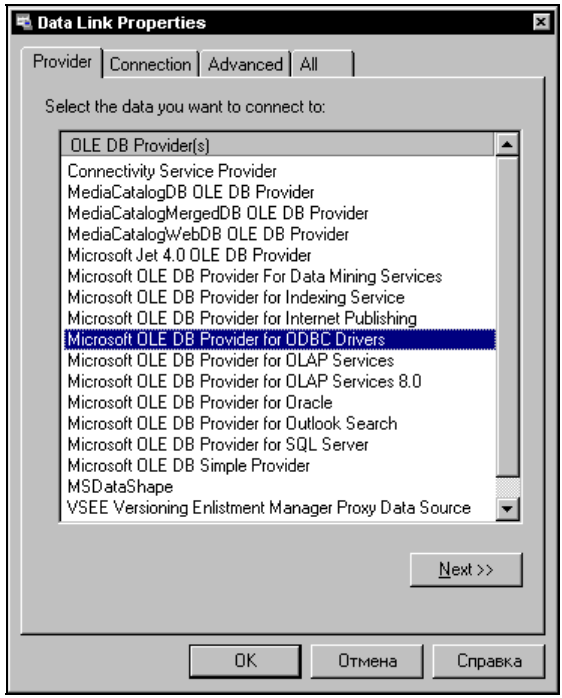


Рис. 20.4. Вкладка **Provider** окна настройки соединения

- **Microsoft Data Shaping Service for OLE DB** — доступ к иерархическим наборам данных;
- **Microsoft OLE DB Simple Provider** — доступ к хранилищам данных, поддерживающим основные возможности OLE DB;
- **Microsoft OLE DB Provider for ODBC drivers** — доступ к данным для драйверов ODBC;
- **Microsoft OLE DB Provider for Oracle** — соединение с сервером Oracle;
- **Microsoft OLE DB Provider for SQL Server** — соединение с сервером Microsoft SQL Server.

При нажатии кнопки **Next>>** (рис. 20.4) происходит переход на вкладку **Connection**, содержимое которой несколько изменяется в зависимости от выбора провайдера. К примеру, вид вкладки **Connection** диалогового окна **Data Link Properties** для случая выбора провайдера **Microsoft Jet 4.0 OLE DB Provider** приведен на рис. 20.5.

На вкладке **Connection** можно указать имя базы данных, имя пользователя и пароль (для защищенных БД). Кроме того, нажав кнопку **Test Connection**, можно проверить правильность функционирования соединения. Далее можно нажатием

кнопки **OK** установить строку соединения, либо перейти на остальные две вкладки.

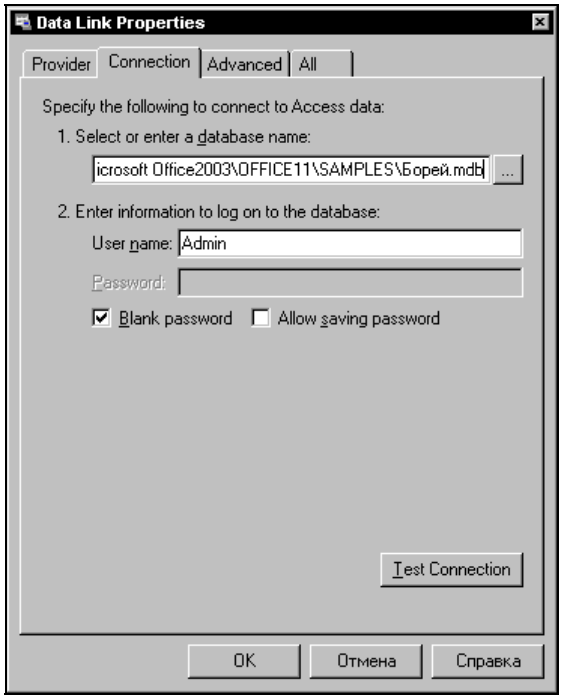


Рис. 20.5. Вкладка **Connection** окна настройки соединения

На вкладке **Advanced** в поле **Network Settings** задается уровень защиты при сетевом доступе к базе данных. В поле **Connect timeout** задается предельное время ожидания соединения в секундах. В списке **Access permissions** для определения прав доступа задается перечень допустимых операций: **Read** — только чтение; **ReadWrite** — чтение и запись; **Share Deny None** — нет запрета на чтение и запись; **Share Deny Read** — запрещено открытие для чтения; **Share Deny Write** — запрещено открытие для записи; **Share Exclusive** — эксклюзивное (монопольное) использование; **Write** — только запись.

На вкладке **All** диалогового окна настройки можно просмотреть и отредактировать параметры соединения, заданные с помощью других вкладок.

В случае использования компонента `ADOConnection` для активизации соединения после настройки достаточно установить свойству `Connected` этого компонента значение `True` или при выполнении приложения вызвать метод `Open`.

В случае использования любого из компонентов доступа к данным (`ADODataSet`, `ADOTable`, `ADOQuery` и `ADOStoredProc`) для активизации соединения после настройки используют свойство `Active`.

Управление соединением и транзакциями

Как следует из изложенного, установление соединения с хранилищем данных может осуществляться компонентами доступа к данным ADO через их свойство `ConnectionString`.

Кроме того, компоненты доступа к данным ADO могут соединяться с хранилищем данных через свойство `Connection`, связывающее их с компонентом соединения `ADOConnection`. При этом последний компонент связывается с хранилищем данных через свое свойство `ConnectionString`.

Достоинство применения компонента `ADOConnection` для связи компонентов доступа к данным с хранилищами данных состоит в том, что он позволяет управлять параметрами соединения и транзакциями.

С помощью свойства `CoursorLocation` типа `TCursorLocation` можно задать библиотеку, используемую курсором при соединении с хранилищем данных. Используемое по умолчанию значение `clUseClient` означает, что все данные располагаются и обрабатываются на компьютере-клиенте. При этом достигается наибольшая гибкость: возможны операции, которые могут не поддерживаться сервером. Тем не менее операторы SQL все равно выполняются на сервере, а клиенту передаются результаты выполнения запроса.

Если свойству `CoursorLocation` задано значение `clUseServer`, то обработка данных ведется на сервере. Такое значение свойства целесообразно устанавливать в случае задания команд, возвращающих большой объем данных.

При использовании технологии dgGo.NET (ADO) соединение может быть *синхронным* или *асинхронным*, что можно задать с помощью свойства `ConnectOptions` типа `TConnectOption`, который описан так:

```
type TConnectOption = (coConnectUnspecified, coAsyncConnect);
```

Значение `coConnectUnspecified` задает синхронное соединение, которое всегда ожидает результат последнего запроса, значение `coAsyncConnect` задает асинхронное соединение, при котором новые запросы выполняются, не ожидая ответа от предыдущих запросов.

Перед установлением соединения возникает событие `OnWillConnect` типа `TWillConnectEvent`, который описан так:

```
TWillConnectEvent = procedure(Connection: TADOConnection; var
 ConnectionString, UserID, Password: WideString; var ConnectOptions:
 TConnectOption; var EventStatus: TEventStatus) of object;
```

Здесь `Connection` указывает на соответствующий компонент соединения; `ConnectionString`, `UserID`, `Password` определяют строку соединения, имя и пароль пользователя соответственно; `ConnectOptions` указывает на синхронное или асинхронное соединение.

Параметр EventStatus указывает на успешность выполнения запроса на соединение. Тип этого параметра описан так:

```
type TEventStatus =
(esOK,esErrorsOccured,esCantDeny,esCancel,esUnwantedEvent);
```

Здесь, например, esOK — соединение выполнено успешно; esErrorsOccured — ошибка при выполнении операции; esCantDeny — незаконченную операцию соединения нельзя отменить.

После установки соединения возникает событие OnConnectComplete типа TConnectErrorEvent, который описан так:

```
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
  var EventStatus: TEventStatus) of object;
```

В случае возникновения ошибки в процессе соединения параметр EventStatus будет иметь значение esErrorsOccured, а параметр Error будет содержать объект ошибки ADO.

Замечание

Объекты ошибки ADO содержат информацию об ошибке, возникшей при выполнении операции над каким-либо объектом ADO. Разработчик может использовать методы объекта ошибки интерфейса Error, предоставляемого многими методами других объектов ADO. Укажем важные свойства объекта ошибки. Свойство Description возвращает описание ошибки, переданное из объекта. Свойство SQLState содержит текст команды, вызвавшей ошибку. Свойство NativeError возвращает код ошибки, переданный из объекта, в котором ошибка произошла.

При разрыве соединения возникает событие OnDisconnect типа TDisconnectEvent, который описан так:

```
TDisconnectEvent = procedure(Connection: TADOConnection; var EventStatus:
 TEventStatus) of object;
```

Управление транзакциями осуществляется с помощью методов и свойств компонента ADOConnection.

Для *запуска* транзакции используется функция BeginTrans, возвращающая целое значение — уровень вложенности новой транзакции. В случае успешного запуска транзакции свойство InTransaction принимает значение True, указывающее на то, что компонент соединения находится в транзакции.

Для *завершения* транзакции и сохранения внесенных изменений в хранилище данных используется метод CommitTrans. При успешном его выполнении свойство InTransaction принимает значение False.

Для *отката* транзакции служит метод RollbackTrans. При его успешном выполнении отменяются все изменения, внесенные в ходе транзакции, а свойство InTransaction принимает значение True.

Для управления запуском транзакций, оставшихся незавершенными, служит свойство Attributes типа TXactAttributes. Оно принимает два значения:

xaCommitRetaining — незавершенная транзакция начинается при подтверждении предыдущей транзакции; xaAbortRetaining — незавершенная транзакция начинается при отмене предыдущей транзакции.

Компоненты доступа к данным

Перечень и назначение компонентов работы по технологии dgGo.NET (ADO) мы приводили в начале главы. Большинство из них имеет аналоги компонентов в технологиях BDE.NET и dbExpress.NET — см. табл. 20.1 (в этой таблице компоненты, используемые в технологии dbExpress.NET, аналогами можно назвать достаточно условно, т. к. большинство из них поддерживают однонаправленные наборы данных).

Таблица 20.1. *Соответствие компонентов dgGo.NET, dbExpress.NET и BDE.NET*

Компонент dgGo.NET (ADO)	Компонент dbExpress.NET	Компонент BDE.NET
ADOTable	SQLTable	Table
ADOQuery	SQLQuery	Query
ADOStoredProc	SQLStoredProc	StoredProc
ADOConnection	SQLConnection	Database
ADODataSet	SQLDataSet, SQLSimpleDataSet	Table, Query, StoredProc
ADOCommand	–	–
RDSConnection	–	–

Стандартные компоненты доступа к данным в dgGo.NET (ADO) наследуют механизм доступа от родительского класса TCustomADODataSet. Поэтому важнейшие свойства и методы этого класса во многом определяют поведение компонентов доступа к данным в dgGo.NET (ADO). Коротко охарактеризуем их.

К числу основных свойств названного класса можно отнести свойства, устанавливающие параметры обмена с хранилищем данных. Значения их нужно задать до открытия набора данных.

Тип блокировки записей в наборе данных определяет свойство LockType типа TADOLockType, который описан так:

```
type TADOLockType = (ltUnspecified, ltReadOnly, ltPessimistic, ltOptimistic,
ltBatchOptimistic);
```

Здесь, например: ltUnspecified — блокировка задается источником данных; ltReadOnly — набор данных открывается в режиме только для чтения; ltPessimistic — блокировка на время редактирования до момента подтверждения.

Как отмечалось, местоположение курсора задает свойство `CursorLocation` (см. предыдущий подраздел). При работе с клиентским курсором важную роль играет следующее свойство.

Передаваемые серверу данные определяет свойство `MarshalOptions` типа `TMarshalOption`, который описан так:

```
type TMarshalOption = (moMarshalAll, moMarshalModifiedOnly);
```

Здесь `moMarshalAll` — возврат всех записей локального набора данных серверу; `moMarshalModifiedOnly` — возврат только измененных записей.

Тип курсора определяет свойство `CursorType` типа `TCursorType`, который описан так:

```
type TCursorType = (ctUnspecified, ctOpenForwardOnly, ctKeyset, ctDynamic,
ctStatic);
```

Здесь, например: `ctUnspecified` — тип курсора не задан и определяется возможностями источника данных; `ctOpenForwardOnly` — однонаправленный курсор, используемый для одиночного прохода по всем записям; `ctKeyset` — двунаправленный курсор, не отображающий добавленные или удаленные другими пользователями записи; `ctStatic` — двунаправленный курсор, не учитывающий изменения записей другими пользователями.

Набор данных может быть открыт с помощью свойства `Active` или метода `Open`. На стороне клиента записи из набора данных хранятся в буфере, размер которого может быть получен с помощью свойства `CacheSize` типа `Integer`.

При необходимости с помощью свойства `BlockReadSize` типа `Integer` можно организовать передачу записей в виде блоков. Кроме того, с помощью свойства `MaxRecords` типа `Integer` можно ограничить размер набора данных. По умолчанию блочная пересылка не используется, а число записей в наборе данных не ограничено.

Текущее *состояние записи* набора данных определяет свойство `RecordStatus` типа `TRecordStatusSet`, который описан так:

```
type TRecordStatus = (rsOK, rsNew, rsModified, rsDeleted, rsUnmodified,
  rsInvalid, rsMultipleChanges, rsPendingChanges, rsCanceled,
  rsCantRelease, rsConcurrencyViolation, rsIntegrityViolation,
  rsMaxChangesExceeded, rsObjectOpen, rsOutOfMemory, rsPermissionDenied,
  rsSchemaViolation, rsDBDeleted);
```

Здесь, например: `rsOK` — запись успешно изменена; `rsNew` — новая запись вставлена; `rsModified` — запись изменена; `rsDeleted` — запись удалена; `rsUnmodified` — запись осталась без изменений.

Дадим сравнительно развернутую характеристику компонентам, используемым в технологии dgGo.NET (ADO) для доступа к данным.

Доступ к таблицам

Для обеспечения доступа к таблицам хранилищ данных по технологии dgGo.NET (ADO) служит компонент `ADOTable`. Для установления соединения с хранилищем данных этого компонента через провайдеры dgGo.NET (ADO) служит свойство `ConnectionString` или `Connection`, как описано ранее. Для управления набором данных таблицы в приложение включают компонент источника данных `DataSource`. При этом свойству `DataSet` этого компонента в качестве значения задается имя компонента `ADOTable`. Для отображения данных таблицы к источнику данных подключаются различные компоненты отображения, к примеру, `DBGrid`.

После установления связи компонента `ADOTable` с хранилищем данных с помощью свойства `TableName` типа `WideString` задается имя таблицы. Не все провайдеры ADO допускают непосредственный доступ к таблицам, поэтому может потребоваться доступ с помощью SQL-запроса. Вариант доступа к данным таблицы определяет свойство `TableDirect` типа `Boolean`. Если оно имеет значение `False` (по умолчанию), то компонент `ADOTable` автоматически генерирует SQL-запрос для доступа к таблице, в противном случае выполняется непосредственный доступ к данным таблицы.

Рассматриваемый нами компонент по своим возможностям и технике работы с ним во многом схож с компонентом `Table`. Здесь также с помощью редактора полей можно задавать свойства отдельных полей. При этом имеется ограничение, состоящее в том, что в компонентах dgGo.NET (ADO) нельзя работать со словарями. Поэтому свойства полей требуется задавать вручную. Кроме того, у драйверов dgGo.NET (ADO) имеются ограничения по работе с отдельными типами полей, в частности с графическими.

Для программирования действий по работе с хранилищем данных с помощью рассматриваемого компонента используются аналогичные средства, что и в случае компонента `Table`. В частности, для навигации по табличному набору данных применяются методы `First`, `Last`, `Next` и `Prior`. Для поиска записей используются методы `Find`, `Seek` и `Locate`.

Выполнение запросов

Для выполнения SQL-запросов при использовании технологии dgGo.NET (ADO) служит компонент `ADOQuery`. По функциональным возможностям и технике применения этот компонент во многом подобен компоненту `Query`. Установка соединения с хранилищем данных, свойства и методы фильтрации и поиска аналогичны используемым для компонента `ADOTable`.

Текст запроса задается с помощью свойства `SQL` типа `TStrings`.

С помощью свойства `Parameters` типа `TParameters` определяются параметры запроса.

Открытие набора данных может быть выполнено с помощью свойства `Active` типа `Boolean` или с помощью метода `Open`. Если же запрос не должен возвращать данных, то для открытия набора данных можно вызвать метод `ExecSQL`.

Вызов хранимых процедур

Для вызова хранимых процедур по технологии dgGo.NET (ADO) служит компонент `ADOStoredProc`. По своим возможностям и технике использования он подобен своему аналогу из BDE.NET. Установка соединения с хранилищем данных и управление набором данных аналогичны используемым для компонента `ADOTable`. Для отображения данных (выходных параметров) к источнику данных также подключаются компоненты отображения.

Для задания имени хранимой процедуры служит свойство `ProcedureName` типа `WideString`.

Свойство `Parameters` типа `TParameters` служит для определения входных параметров процедуры. А именно, после задания имени хранимой процедуры в свойстве `Parameters` отображаются входные параметры процедуры. Их можно просмотреть с помощью Инспектора объектов.

Выходные параметры хранимой процедуры являются объектами полей рассматриваемого компонента `ADOStoredProc`. Чтобы просмотреть и изменить их значения, достаточно с помощью контекстного меню компонента `ADOStoredProc` вызвать редактор полей Fields Editor. С помощью команд редактора можно легко изменить состав полей (выходные параметры), отображаемых в наборе данных компонента `ADOStoredProc`.

Компонент *ADODataSet*

Компонент `ADODataSet` служит для представления набора данных из хранилища данных dgGo.NET (ADO). По своим функциональным возможностям компонент в определенной мере аналогичен компонентам `SQLDataSet` и `SQLSimpleDataSet`, используемым в технологии dbExpress. Этот компонент позволяет получать данные из таблиц, SQL-запросов, хранимых процедур, файлов и т. д.

Тип команды, указываемой в свойстве `CommandText`, определяет свойство `CommandType` типа `TCommandType`, который определен так:

```
TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc, cmdFile,
   cmdTableDirect);
```

Здесь `cmdUnknown` — тип не известен; `cmdText` — текст SQL-оператора или вызова процедуры (значение по умолчанию); `cmdTable` — имя таблицы; `cmdStoredProc` — имя хранимой процедуры; `cmdFile` — имя сохраненного файла набора данных; `cmdTableDirect` — имя таблицы, все поля которой возвращаются.

Значение свойства `CommandText` типа `WideString` определяет команду, которая выполняется. В качестве команды может быть задана строка с SQL-оператором, имя

таблицы или имя хранимой процедуры. При необходимости могут быть заданы параметры с помощью свойства Parametrs. Выполнение команды, заданной с помощью свойства CommandText, происходит при открытии набора данных.

Установка соединения с хранилищем данных и управление набором данных аналогичны используемым для компонента ADOTable. Для отображения данных к источнику данных (компоненту DataSource) также подключаются компоненты отображения.

Команды

Для выполнения команд предназначен компонент ADOCommand, являющийся реализацией объекта ADO Command в среде Delphi .NET 2006. Этот компонент служит для выполнения команд, не возвращающих результаты, например, SQL-операторов языка определения данных или хранимых процедур. Для выполнения хранимых процедур и SQL-запросов, возвращающих результаты в виде наборов данных, целесообразно применять компоненты доступа к данным ADODataSet, ADOQuery и ADOStoredProc.

Рассмотрим важнейшие свойства и методы компонента ADOCommand. Отметим, что большинство свойств компонента ADOCommand в Delphi .NET 2006 эквивалентны свойствам объекта ADO Command.

Свойство CommandObject типа _Command предоставляет непосредственный доступ к базовому объекту команды ADO Command.

Тип команды, указываемой в свойстве CommandText, определяет свойство CommandType (см. описание компонента ADODataSet). Для свойства CommandType объекта ADOCommand значения cmdFile и cmdTableDirect являются недопустимыми.

Напомним, что значение свойства CommandText типа WideString определяет команду, которая выполняется с помощью метода Execute. В качестве команды может быть задана строка с SQL-оператором, имя таблицы или имя хранимой процедуры. При необходимости могут быть заданы параметры с помощью свойства Parametrs.

Текущее состояние компонента ADOCommand (объекта Command) определяет свойство States типа ObjectStates, который описан так:

```
TObjectState = (stClosed, stOpen, stConnecting, stExecuting, stFetching);
```

Здесь stClosed — объект Command не активен и не соединен с хранилищем данных; stOpen — объект Command не активен, соединен с хранилищем данных; stConnecting — объект Command соединяется с хранилищем данных; stExecuting — объект Command выполняет команду; stFetching — объект Command получает данные от хранилища данных.

Условия выполнения команды ADO определяет свойство ExecuteOptions типа TExecuteOptions, который описан так:

```
TExecuteOption = (eoAsyncExecute, eoAsyncFetch, eoAsyncFetchNonBlocking,
  eoExecuteNoRecords);
```

Здесь `eoAsyncExecute` — асинхронное выполнение команды; `eoAsyncFetch` — асинхронная передача данных; `eoAsyncFetchNonBlocking` — асинхронная передача данных без блокирования потока; `eoExecuteNoRecords` — выполнение команды ADO без возвращения данных.

Выполнение команды ADO осуществляется с помощью одной из трех перегружаемых версий метода `Execute` с различными наборами параметров:

- `function Execute: _Recordset; overload;` — такое объявление метода используется при выполнении команды без параметров;
- `function Execute(const Parameters: OleVariant): _Recordset; overload;` — в этом объявлении `Parameters` определяет параметры команды;
- `function Execute(var RecordsAffected: Integer; const Parameters: OleVariant): _RecordSet; overload;` — здесь параметр `RecordsAffected` возвращает общее число обработанных запросом записей.

Пример приложения

Рассмотрим пример (рис. 20.6) приложения для работы с БД, использующего технологию dgGo.NET (ADO). Пусть назначением приложения является организация доступа к различным таблицам из базы данных, их редактирование и сохранение внесенных изменений на сервере.

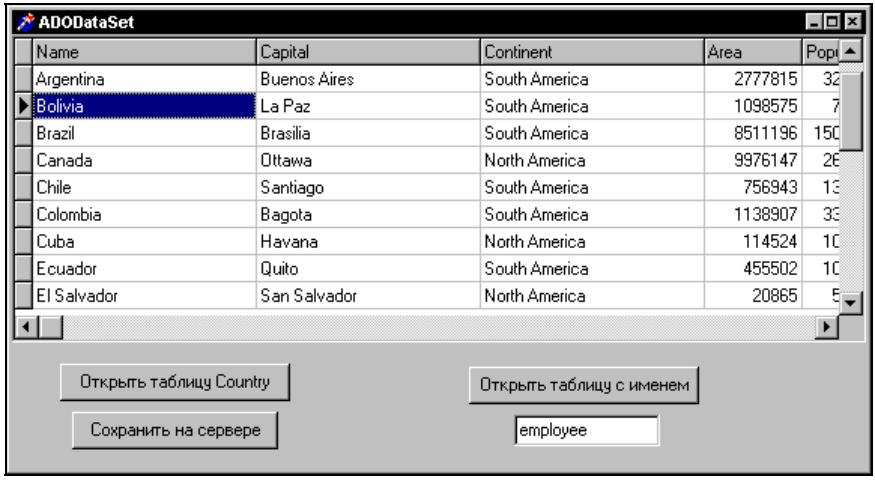


Рис. 20.6. Вид формы при выполнении приложения

Основными компонентами приложения, используемыми для установления соединения с базой данных и отображения данных, являются `ADOConnection1`,

ADODataSet1, DataSource1 и DBGrid1. Кроме того, имеется ряд вспомогательных компонентов, служащих для управления работой приложения.

Соединение с хранилищем данных компонента ADODataSet1 выполнено через его свойство Connection, указывающее на имя ADOConnection1 компонента соединения. Для компонента ADOConnection1 с помощью его свойства ConnectionString установлено соединение с хранилищем данных. При этом использован файл связи с данными. Для наглядности в процедуре обработки события создания формы приведен оператор задания нужного значения свойству ConnectionString компонента ADOConnection1.

Для компонента DataSource1 его свойству DataSet установлено значение ADODataSet1. Для компонента DBGrid1, используемого для отображения набора данных, его свойству DataSource установлено значение DataSource1.

Код модуля формы для решения поставленной задачи имеет следующий вид:

```
unit ADODataSet;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, DB, ADODB, Grids, DBGrids, ExtCtrls;

type
  TForm1 = class(TForm)
    ADODataSet1: TADODataSet;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    ADOConnection1: TADOConnection;
    Panel1: TPanel;
    UpdateButton: TButton;
    GetTableButton: TButton;
    Button1: TButton;
    Edit1: TEdit;
    procedure Form1Create(Sender: TObject);
    procedure Form1CloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure UpdateButtonClick(Sender: TObject);
    procedure GetTableButtonClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    private
public
    procedure UpdateData;
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.dfm}

const
  BaseFileName = 'EMPLOYEE.ADTG';

procedure TForm1.UpdateData;
begin
  { Обновление данных на сервере }
  ADODataSet1.UpdateBatch;
end;

procedure TForm1.Form1Create(Sender: TObject);
begin
  ADOConnection1.ConnectionString := 'FILE NAME=' + DataLinkDir +
'\DBDEMOS.UDL';
  ADODataSet1.Open;
end;

procedure TForm1.UpdateButtonClick(Sender: TObject);
begin
  UpdateData;
end;

procedure TForm1.Form1CloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if ADODataSet1.Active then
  try
    { Обновление данных на сервере при закрытии формы }
    UpdateData
  except
    on E: Exception do
    begin
      Application.HandleException(Self);
      CanClose := MessageDlg('Данные не сохранены/обновлены, выйти?',
        mtConfirmation, mbYesNoCancel, 0) = mrYes;
    end;
  end;
end;

procedure TForm1.GetTableButtonClick(Sender: TObject);
begin
  { Повторное открытие табличного набора данных'Country'}
  ADODataSet1.Close;
  ADODataSet1.CommandType := cmdTable;
```

```
  ADODataSet1.CommandText := 'Country';
  ADODataSet1.Open;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ADODataSet1.Close;
  ADODataSet1.CommandType := cmdTable;
  ADODataSet1.CommandText := Edit1.Text;
  try
      ADODataSet1.Open;
  except
    on E: Exception do
    begin
      MessageDlg('Исправьте имя таблицы', mtError,[mbOK],0);
      ADODataSet1.Close;
    end;
end;
end;
end.
```

Первоначально при запуске приложения автоматически открывается таблица с именем `Country`. При необходимости открыть другую таблицу для просмотра и редактирования, ее имя следует ввести в однострочном редакторе (компонент `Edit1`) и затем нажать кнопку **Открыть таблицу с именем**.

Обновление внесенных в открытую таблицу изменений происходит при нажатии кнопки **Сохранить на сервере** или автоматически при закрытии формы приложения.

ГЛАВА 21

Технология dbExpress.NET

Общая характеристика

В основе технологии dbExpress.NET лежит использование множества легковесных драйверов, компонентов, объединяющих соединения, транзакции, запросы и наборы данных, а также интерфейсов, реализующих универсальный доступ к соответствующим функциям.

По сравнению с использованием механизма BDE.NET технология dbExpress.NET обеспечивает построение более легковесных (по объему кода) приложений для работы с базами данных. При ее применении для доступа к данным используются SQL-запросы. Технология dbExpress.NET обеспечивает легкую переносимость приложений, допускает работу приложений баз данных под управлением Windows и Linux.

Для использования технологии dbExpress.NET достаточно включить в распространяемое приложение динамически подключаемую библиотеку с драйвером, взаимодействующим с клиентским программным обеспечением для нужного сервера базы данных (рис. 21.1).

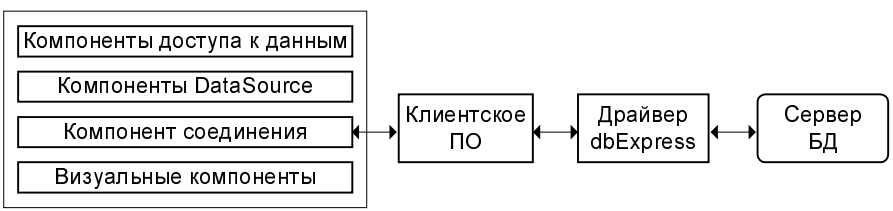


Рис. 21.1. Схема доступа к данным с помощью dbExpress.NET

Драйверы, используемые для доступа к различным серверам баз данных по технологии dbExpress.NET, реализованы в виде динамически подключаемых биб-

лиотек (DLL) (см. рис. 21.3). Драйверы dbExpress.NET в составе Delphi .NET размещаются в папке C:\Program Files\Borland\BDS\4.0\BIN\.

На странице **dbExpress** Палитры инструментов Delphi .NET находятся компоненты, используемые в технологии dbExpress.NET: `SQLConnection` (`Database`), `SQLDataSet`, `SQLQuery` (`Query`), `SQLStoredProc` (`StoredProc`), `SQLTable` (`Table`), `SQLMonitor` (утилита SQL Monitor) и `SimpleDataSet` (`BDEClientDataSet`). Для наглядности в круглых скобках указаны аналоги этих компонентов, используемые в случае механизма BDE.NET. Как видим из приведенного списка, у компонента `SQLDataSet` нет аналога для механизма BDE.NET. Кроме того, ряд компонентов из механизма BDE.NET не имеет аналогов для технологии dbExpress.NET.

Определенным недостатком технологии dbExpress.NET является то, что несколько из перечисленных компонентов (`SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`) являются однонаправленными наборами данных, в которых отсутствует буферизация. Эти наборы данных обеспечивают более быстрый доступ к данным и предъявляют меньшие требования к ресурсам, но при этом на них накладываются заметные ограничения. Для компонента `SimpleDataSet` большинство из этих ограничений не действует.

Установление соединения с сервером

Для установления соединения с сервером базы данных служит компонент `SQLConnection`, который представляет собой аналог компонента `DataBase` в BDE.NET.

Этот компонент взаимодействует с двумя файлами, расположенными в каталоге ..\Common Files\Borland Shared\DBExpress. Файл dbxdrivers.ini содержит список инсталлированных драйверов серверов БД и для каждого драйвера список динамически подключаемых библиотек и параметров соединений, установленных по умолчанию. Список соединений с параметрами соединений содержится в файле dbxconnections.ini.

Поместив компонент `SQLConnection` на форму (или модуль данных) на этапе разработки приложения, можно выбрать одно из существующих соединений, либо создать новое соединение с помощью диалогового окна Редактора соединений (рис. 21.2). Вызвать указанное окно можно, выбрав пункт **Edit Connection Properties** контекстного меню компонента.

Редактор соединений позволяет настроить существующее соединение dbExpress или создать новое соединение, а также проверить правильность настроек с помощью кнопки **Test Connection**. Параметры соединения можно также настраивать с помощью Инспектора объектов (свойство `Params` типа `TStrings`).

Нажав кнопку **View Driver Settings**, можно вызвать диалоговое окно **dbExpress Drivers** (рис. 21.3), в котором указаны имена драйверов, имена библиотек DLL и библиотеки поставщиков.

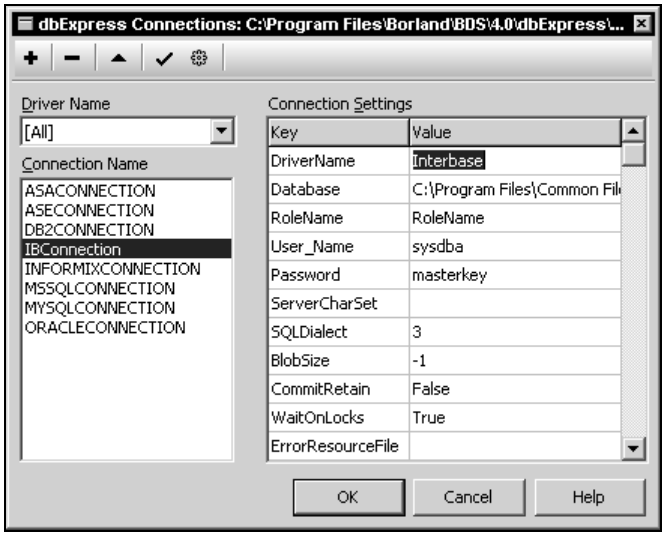


Рис. 21.2. Диалоговое окно Редактора соединений

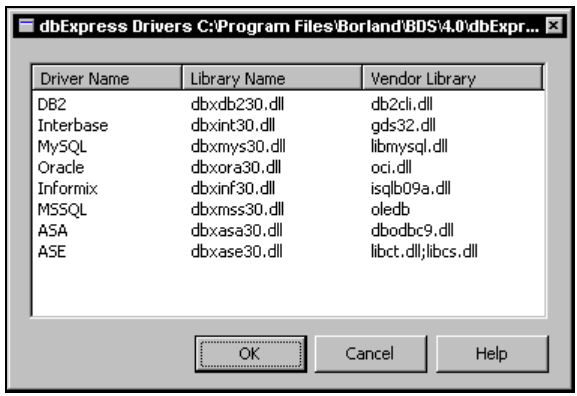


Рис. 21.3. Список драйверов dbExpress

Параметры соединения для основных серверов баз данных приведены в табл. 21.1.

Таблица 21.1. *Параметры соединений для серверов БД*

Сервер БД	Параметр	Значение
Все серверы	`BlobSize`	Ограничение на объем пакета для данных типа `BLOB`
	`DriverName`	Имя драйвера
	`ErrorResourceFile`	Файл сообщений об ошибках

Таблица 21.1 *(окончание)*

Сервер БД	Параметр	Значение
	LocaleCode	Код локализации, определяющий влияние национальных символов на сортировку данных
	User_Name	Имя пользователя
	Password	Пароль
	<имя сервера БД> TransIsolation	Уровень изоляции транзакций для сервера с заданным именем
InterBase	CommitRetain	Поведение курсора по завершению транзакции: True — обновление; False — удаление
	RoleName	Роль пользователя
	SQLDialect	Используемый диалект SQL
	Trim Char	Возможность удаления из строковых значений полей пробелов, дополняющих их до полной строки
	WaitOnLocks	Разрешение на ожидание занятых ресурсов
	DataBase	Имя файла базы данных (с расширением gdb)
DB2	DataBase	Имя клиентского ядра
MySQL	DataBase	Имя базы данных
Microsoft SQL Server 2000, MySQL, Informix	HostName	Имя компьютера, на котором установлен сервер
Microsoft SQL Server 2000, Informix	DataBase	Псевдоним (алиас) имени базы данных
Microsoft SQL Server 2000, Oracle	OS Autenification	Использование учетной записи текущего пользователя операционной системы при доступе к ресурсам сервера
Informix, Oracle	Trim Char	Возможность удаления из строковых значений полей пробелов, дополняющих значение до полной строки
Oracle	AutoCommit	Флаг завершения транзакции. Устанавливается сервером
	BlockingMode	Режим завершения запроса. True — соединение дожидается окончания запроса; False — начинает выполнение следующего запроса
	Multiple Transaction	Возможность управления несколькими транзакциями в одном сеансе
	DataBase	Запись базы данных в файле TNSNames.ora

После описанной настройки параметров соединений для компонента SQLConnection с помощью Инспектора объектов можно выбрать соединение (свойство ConnectionName) для нужного сервера баз данных. При этом автоматически устанавливаются значения связанных с ним свойств: DriverName (имя драйвера); LibraryName (имя динамически подключаемой библиотеки драйвера); Params (параметры соединения) и VendorLib (имя динамически подключаемой библиотеки клиентской части сервера).

Открытие соединения с сервером БД осуществляется заданием свойству Connected типа boolean значения True. Например, при выполнении приложения это можно сделать с помощью кода:

```
SQLConnection1.Connected := True;
```

Соответственно закрыть соединение с сервером можно путем задания этому свойству значения False. Названные операции могут быть выполнены также с помощью методов:

```
procedure Open;
procedure Close;
```

С помощью свойства LoginPrompt типа boolean выполняется задание необходимости (True) или ненужности (False) отображения окна авторизации для ввода имени пользователя (User_Name) и пароля (Password) при каждой попытке соединения с сервером. В последнем случае эти значения будут браться из файла dbxconnections.ini.

После открытия соединения все компоненты dbExpress.NET, инкапсулирующие наборы данных и связанные с открытым компонентом SQLConnection, получают доступ к базе данных.

При открытии и закрытии соединения можно воспользоваться любым из доступных событий, например:

```
property AfterConnect: TNotifyEvent;
property AfterDisconnect: TNotifyEvent;
property BeforeConnect: TNotifyEvent;
property BeforeDisconnect: TNotifyEvent;
property OnLogin: TSQLConnectionLoginEvent;
```

Последнее событие наступает в случае, если свойству LoginPrompt установлено значение True. Если это свойство имеет значение True, и нет обработки события OnLogin, то пользователю будет предложен стандартный диалог авторизации.

Узнать о *текущем состоянии соединения* можно, проанализировав свойство ConnectionState типа TConnectionState, тип которого описан так:

```
TConnectionState = (csStateClosed, csStateOpen, csStateConnecting,
 csStateExecuting, csStateFetching, csStateDisconnecting);
```

Указанные в определении типа значения соответствуют следующим состояниям соединения: закрыто, открыто, установление соединения, ожидание исполнения переданного SQL-запроса, получение данных с сервера, завершение соединения.

Компонент `SQLConnection` позволяет выполнять SQL-запросы с помощью следующих функций.

Функция `Execute(const SQL: string; Params: TParams; ResultSet: Pointer = Nil): Integer;` выполняет запрос, определенный значением константы `SQL`, с параметрами `Params`, используемыми в SQL-запросе. Используемые в запросе параметры должны иметь тот же порядок следования, что и в `Params` и в инструкции SQL. Если SQL-запрос не содержит параметров, то свойству `Params` требуется задать значение `Nil`. Если при выполнении SQL-запроса возвращается результат, то в параметр `ResultSet` помещается указатель на объект типа `TCustomSQLDataSet`, содержащий результат.

При отсутствии в запросе параметров и возвращения записей рекомендуется использовать функцию `ExecuteDirect(const SQL: string): Integer`, которая возвращает значение 0 при успешном завершении запроса или код ошибки в противном случае.

Подобно своим аналогам в BDE.NET компонент `SQLConnection` позволяет выполнять запуск, фиксацию и откат *транзакций* соответственно с помощью методов: `StartTransaction, Commit` и `Rollback`.

Компоненты доступа к данным

Как отмечалось, ряд компонентов доступа к данным по технологии dbExpress.NET (`SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`) являются однонаправленными наборами данных, в которых отсутствует буферизация и при этом на них накладываются заметные ограничения.

В однонаправленных наборах данных используются однонаправленные курсоры. С их помощью допускается только получать данные, из методов навигации по набору данных поддерживаются лишь методы `First` и `Next`, подразумевающие последовательный перебор записей от начала к концу.

При использовании однонаправленных наборов данных отсутствует возможность прямого редактирования данных, т. к. для этого нужно размещение в буфере результатов редактирования. При этом соответствующее свойство названных компонентов (`CanModify`) всегда имеет значение `False`. Для однонаправленных наборов данных возможности редактирования данных все же доступны, например, путем задания оператора `UPDATE` языка SQL. Кроме того, для однонаправленных наборов данных по тем же причинам имеют место ограничения по выполнению фильтрации. Это ограничение также можно обойти путем указания параметров фильтрации в SQL-запросах.

Имеются также ограничения по отображению данных с помощью компонентов страницы **Data Controls**. В частности, нельзя использовать компоненты `DBGrid` и `DBCtrlGrid`, а также компоненты синхронного просмотра. Для компонента навигатора требуется отключить кнопки возврата на последнюю запись и перехода на последнюю запись. Остальные компоненты можно использовать как обычно.

Для компонента `SimpleDataSet` большинство из указанных ограничений не действует. Он использует внутренние компоненты типа `TSQLDataSet` и `TDataSetProvider` для получения и изменения данных.

Перед использованием любого из указанных компонентов доступа к данным следует подключить к серверу БД. Для этого достаточно его свойству `SQLConnection` в качестве значения установить имя компонента соединения `SQLConection`.

Компоненты доступа к данным можно использовать, поместив на форме, либо динамически — создав при выполнении приложения.

Универсальный доступ к данным

В приложениях VCL.NET для обеспечения универсального однонаправленного доступа к данным БД по технологии dbExpress.NET служит компонент `SQLDataSet`. Он позволяет получать все записи из таблицы БД, производить выборку данных путем выполнения SQL-запросов или выполнять хранимые процедуры.

Для определения типа выполняемых действий (с запросами, таблицами или хранимыми процедурами) свойству `CommandType` рассматриваемого компонента нужно установить соответственно одно из трех возможных значений:

- `ctQuery` — в свойстве `CommandText` указывается SQL-запрос;
- `ctTable` — в свойстве `CommandText` указывается имя таблицы на сервере БД, при этом компонент автоматически генерирует SQL-запрос для получения всех записей для всех полей таблицы;
- `ctStoredProc` — в свойстве `CommandText` указывается имя хранимой процедуры.

В случае выбора значения `ctQuery` текст SQL-запроса можно ввести в поле свойства `CommandText` с помощью Инспектора объектов или редактора построения запроса **CommandText Editor** (рис. 21.4), вызываемого нажатием кнопки с тремя точками в строке этого свойства. В поле **Connection** редактора нужно из списка выбрать имя соединения, а в поле **Schema Name** — имя пользователя, используемое в параметрах этого соединения. Затем нужно нажать кнопку **Get Database Objects**. При этом в списке **Tables** становятся доступны имена таблиц в базе данных, а в списке **Fields** — имена полей выбранной таблицы. С помощью кнопок **Add Table to SQL** и **Add Field to SQL** можно добавить нужные таблицы и поля. При этом в поле **SQL** отображается автоматически сформированный SQL-запрос. После нажатия кнопки **OK** в окне редактора мы получаем нужное значение свойства `CommandText`.

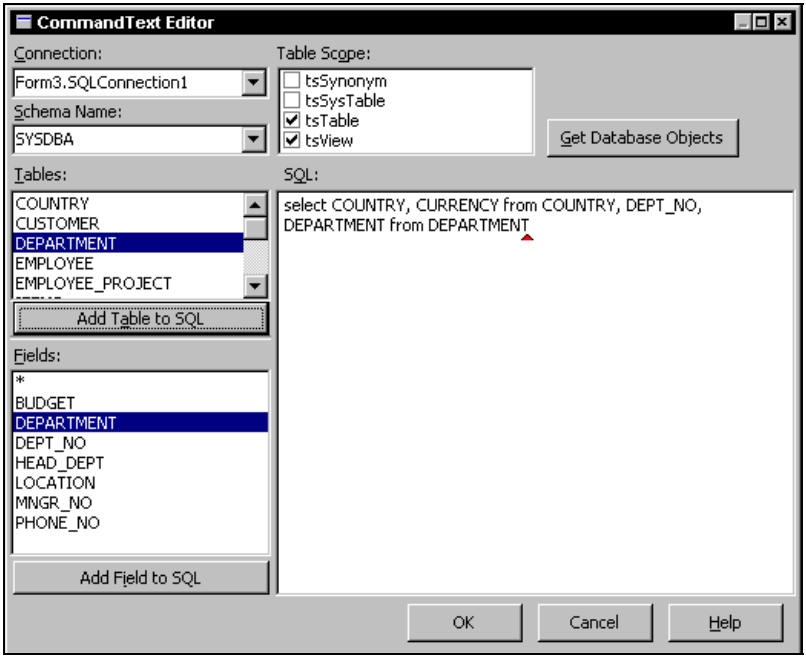


Рис. 21.4. Редактор построения SQL-запроса

При выполнении приложения аналогичные действия (как на рис. 21.4) для выполнения SQL-запроса можно задать так:

```
SQLDataSet1.CommandType := ctQuery;
SQLDataSet1.CommandText := 'select COUNTRY, CURRENCY from COUNTRY, DEPT_NO,
DEPARTMENT from DEPARTMENT ';
SQLDataSet1.ExecSQL;
```

При выборе значения `ctTable` в свойстве `CommandText` указывается просто имя таблицы, например, так:

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'COUNTRY';
SQLDataSet1.ExecSQL;
```

Для открытия набора данных нужно установить значение `True` свойству `Active` или использовать метод `Open`. Если же SQL-запрос или хранимая процедура не возвращают набор данных, для их выполнения используется метод

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Параметр `ExecDirect` определяет необходимость подготовки параметров перед выполнением инструкции. Если параметры запроса или процедуры существуют, параметр `ExecDirect` должен иметь значение `False`.

При необходимости сортировки данных используют инструкцию `ORDER BY` в случае варианта `ctQuery`, либо устанавливают с помощью редактора **SortFields Editor** значение свойства `SortFieldNames` в случае варианта `ctTable`. Для варианта `ctStoredProc` порядок сортировки определяется в хранимой процедуре.

При необходимости в SQL-запросе или в хранимой процедуре можно использовать параметры, позволяющие настраивать запрос или процедуру, не изменяя их код. Параметры задаются с помощью свойства `Params`, которое является коллекцией объектов типа `TParam`.

Свойство `DataSource` типа `TDataSource` компонента `SQLDataSet` позволяет связать два набора данных по схеме "мастер-детальный" *(см. главу 14)*.

Метод `SetSchemaInfo` компонента `SQLDataSet` позволяет получить метаданные (список таблиц на сервере, список системных таблиц, информацию о хранимых процедурах, информацию о полях таблицы, параметры хранимой процедуры).

Рассмотрим пример формы (рис. 21.5) приложения БД, в котором с помощью компонентов `SQLDataSet` выполняется доступ к данным по технологии dbExpress.NET. Пусть требуется обеспечить возможность просмотра и редактирования полей данных `CUST_NO`, `CUSTOMER` и `PHONE_NO` таблицы `CUSTOMER`, принадлежащей базе данных `EMPLOYEE.GDB` сервера InterBase.

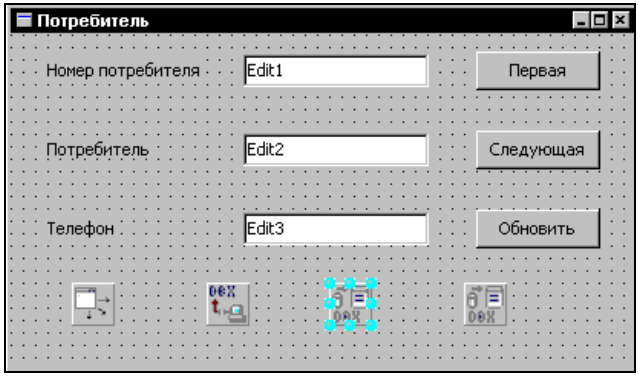


Рис. 21.5. Вид формы при разработке

Код модуля формы для решения поставленной задачи имеет вид:

```
unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DBXpress, FMTBcd, Borland.Vcl.SqlExpr, Borland.Vcl.StdCtrls,
  Borland.Vcl.Db, System.ComponentModel;
```

```
type
  TForm3 = class(TForm)
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
    DataSource1: TDataSource;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    SQLDataSet2: TSQLDataSet;
    procedure FormCreate(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure SQLDataSet1AfterScroll(DataSet: TDataSet);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form3: TForm3;

implementation

{$R *.nfm}

procedure TForm3.Button1Click(Sender: TObject);
begin
SQLDataSet1.First;
end;

procedure TForm3.Button2Click(Sender: TObject);
begin
SQLDataSet1.Next;
end;

procedure TForm3.Button3Click(Sender: TObject);
begin
       with SQLDataSet2 do
```

```
        try
        CommandText := 'UPDATE CUSTOMER SET CUST_NO = :Cust_no, CUSTOMER =
                    :Customer, PHONE_NO = :Phone_no WHERE CUST_NO = :CUST_NO';
        Params[0].AsString := Edit1.Text;
        Params[1].AsString := Edit2.Text;
        Params[2].AsString := Edit3.Text;
        ExecSQL;
        except
          MessageDlg('Customer info fixed error',mtError,[mbOK],0);
        end;
        end;

procedure TForm3.FormCreate(Sender: TObject);
begin
SQLDataSet1.Active:=False;
SQLDataSet1.CommandType:=ctTable;
SQLDataSet1.SQLConnection:=SQLConnection1;
SQLDataSet1.CommandText:='CUSTOMER';
SQLDataSet1.Active:=True;
SQLDataSet2.SQLConnection:=SQLConnection1;
SQLDataSet2.CommandType:=ctQuery;
end;

procedure TForm3.SQLDataSet1AfterScroll(DataSet: TDataSet);
begin
  Edit1.Text:=SQLDataSet1.FieldByName('CUST_NO').AsString;
  Edit2.Text:=SQLDataSet1.FieldByName('CUSTOMER').AsString;
  Edit3.Text:=SQLDataSet1.FieldByName('PHONE_NO').AsString;
end;

end.
```

Для просмотра и редактирования трех указанных полей таблицы `CUSTOMER` используются два компонента `SQLDataSet1` и `SQLDataSet2` соответственно. Оба компонента подсоединены к таблице сервера InterBase через компонент `SQLConnection1`. Первый компонент `SQLDataSet1` служит для просмотра записей таблицы `CUSTOMER`. Второй компонент `SQLDataSet2` служит для фиксации внесенных изменений (три компонента типа `TEdit`) в текущую запись в таблице на сервере БД. В обработчике события `FormCreate` для наглядности выполнены начальные установки важнейших свойств компонентов `SQLDataSet1` и `SQLDataSet2`.

Для заполнения компонентов типа `TEdit` при навигации по набору данных используется обработчик события `AfteScroll` для компонента `SQLDataSet1`. Навигация по набору данных выполняется с помощью обработчиков событий `OnClick` для кнопок `Button1` и `Button2` с заголовками **First** и **Next**. Фиксация внесенных в просматриваемую запись изменений в таблице на сервере осуществляется с по-

мощью SQL-запроса `UPDATE`, размещенного в обработчике события `OnClick` для кнопки `Button3` с заголовком **Update**. В параметрах запроса передаются текущие значения полей из компонентов `Edit1`, `Edit2` и `Edit3` (рис. 21.6).

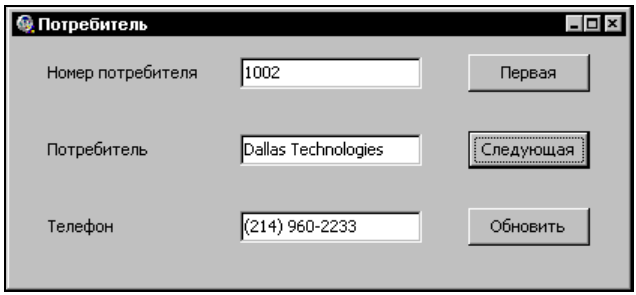


Рис. 21.6. Вид формы при выполнении приложения

Отметим, что необходимость использования компонента `SQLDataSet2` для фиксации внесенных изменений в таблице на сервере вызвана тем, что компонент `SQLDataSet1` является однонаправленным набором данных. Заметим также, что в нашем примере нельзя вместо компонентов `Edit` использовать компоненты `DBEdit`, т. к. здесь они не позволяют выполнять редактирование содержимого полей.

Просмотр таблиц

Для просмотра таблиц по технологии dbExpress.NET служит компонент `SQLTable`. Компонент `SQLTable` генерирует SQL-запрос для получения всех строк и полей указанной таблицы.

Имя таблицы определяется с помощью свойства `TableName`, и при подключении компонента к соединению его можно выбрать в комбинированном списке в окне Инспектора объектов.

Для получения табличного набора данных компонент `SQLTable` с помощью метода

```
procedure PrepareStatement; override;
```

генерирует запрос на сервер БД.

Порядок следования данных в наборе определяется свойством `IndexFieldNames` типа `String` или `IndexName` типа `String`.

Список индексов таблицы можно получить с помощью метода `GetIndexNames(List: TStrings)` в качестве значения параметра `List`.

Связь между двумя наборами данных "главный-подчиненный" устанавливается с помощью свойств `MasterFields и MasterSource`. На этапе разработки приложения двойным щелчком на свойстве `MasterFields` в окне Инспектора объектов можно

открыть диалоговое окно редактора связей для визуального построения отношения "мастер-детальный".

Компонент `SQLTable` является однонаправленным курсором, тем не менее он допускает удаление записей из таблицы на сервер БД с помощью метода `DeleteRecords`.

Выполнение SQL-запроса

Для выполнения SQL-запроса на сервере БД служит компонент `SQLQuery`. Он позволяет представлять результаты выполнения запросов с помощью инструкции `SELECT` или осуществлять действия по изменению БД путем выполнения инструкций `INSERT`, `DELETE`, `UPDATE`, `ALTER TABLE` и т. п.

Для компонента `SQLQuery` текст SQL-запроса определяется как значение свойства `SQL` типа `TStrings`. На этапе разработки приложения VCL.NET текст SQL-запроса может быть набран непосредственно в окне редактора строк (или редактора кода), открываемого двойным щелчком на свойстве `SQL` в окне Инспектора объектов.

При выполнении приложения очистка содержимого свойства `SQL` компонента `SQLQuery` и его модификация могут быть выполнены, к примеру, с помощью следующего кода:

```
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.Add('SELECT ' + Edit1.Text + ' FROM ' + Edit2.Text);
if Length(Edit3.Text) <> 0 then
  SQLQuery1.SQL.Add('ORDER BY ' + Edit3.Text)
```

Свойство `Text` типа `String` в качестве значения содержит строковое представление SQL-запроса в виде одной строки, а не нескольких, как это может быть в случае свойства `SQL`.

Свойство `Active` типа `Boolean` определяет, открыт набор данных или нет. Это свойство обычно используют, чтобы определить или установить заполнение набора данными.

Открыть набор данных SQL-запроса также можно с помощью метода `Open`, либо с помощью метода:

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Здесь значение `False` параметра `ExecDirect` означает, что запрос не имеет настраиваемых параметров.

Свойство `DataSource` связывает набор данных SQL-запроса с другим (главным) набором данных. Этот набор данных используется для получения параметров запроса в случае, когда предусматривается параметризованный запрос, но приложение этими параметрами не обеспечивает.

Выполнение хранимых процедур

Для выполнения хранимых процедур, размещенных на сервере БД, служит компонент `SQLStoredProc`.

Имя хранимой процедуры задает свойство `StoredProcName` типа `String`. Для задания параметров хранимой процедуры предназначено свойство `Params` типа `TParams`. При обращении к параметрам хранимой процедуры целесообразно использовать метод `ParamByName`. Это обусловлено тем, что при работе с некоторыми серверами порядок следования параметров до и после выполнения процедуры может меняться.

Для подготовки хранимой процедуры к выполнению на сервере служит метод:

```
PrepareStatement(var RecordsAffected: Integer): TCustomSQLDataSet;
```

При его вызове сервером БД выделяются ресурсы и связываются их параметры. Поименованные параметры временно преобразуются к непоименованным параметрам, поскольку dbExpress.NET поименованные параметры не поддерживает.

Если хранимая процедура не возвращает набор данных, то ее запускают с помощью метода:

```
function ExecProc: Integer; virtual;
```

В противном случае используется метод `Open` либо свойству `Active` устанавливают значение `True`.

Компонент редактирования набора данных

Для редактирования набора данных (получения данных, их кэширования и отправления измененных данных на сервер) предназначен компонент `SimpleDataSet`. Этот компонент использует двунаправленный курсор и позволяет редактировать данные, но в режиме редактирования. Тем самым он исправляет основные недостатки рассматриваемой технологии.

Для подготовки компонента `SimpleDataSet` к работе с данными нужно с помощью свойства `Connection` связать его с компонентом соединения `SQLConnection`. В качестве альтернативы можно с помощью подсвойства `ConnectionName` свойства `Connection` задать тип соединения непосредственно.

Произведенные над данными изменения размещаются в локальном кэше, в связи с этим для подтверждения изменений и отправки данных на сервер БД используют метод:

```
function ApplyUpdates(MaxErrors: Integer); Integer; virtual;
```

Здесь параметр метода определяет число ошибок, допустимых при передаче данных.

Локальный кэш компонента после сохранения изменений на сервере можно очистить от данных с помощью метода:

```
function Reconcile(const Results: OleVariant): Boolean;
```

Отмена локальных изменений данных может быть выполнена с помощью метода:

```
procedure CancelUpdates;
```

Пересылка данных между сервером и рассматриваемым компонентом осуществляется с помощью пакетов. Размер пакета (по числу записей) можно задать с помощью свойства `PacketRecords` типа `Integer`. По умолчанию устанавливается значение `-1`, которое говорит, что один пакет должен содержать все записи набора данных.

Если значение свойства `PacketRecords` больше `0`, то оно определяет число записей, которые можно получить в пакете от провайдера с помощью метода.

```
function GetNextPacket: Integer;
```

Если значение свойства `PacketRecords` равно `0`, то в пакете передаются только метаданные.

Свойство `DataSize` типа `Integer` устанавливает размер (в байтах) текущего пакета, доступного с помощью свойства `Data` типа `OleVariant`.

На общее число записей в источнике данных указывает свойство `RecordCount` типа `Integer`, номер текущей записи определяет свойство `RecNo` типа `Integer`.

Изменения в текущей записи можно отменить с помощью метода:

```
procedure RevertRecord;
```

Отменить последнюю операцию по изменению клиентского источника данных можно с помощью метода:

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

Здесь значение параметра определяет, где будет установлен курсор после восстановления записи: `True` — на восстановленной записи, `False` — на текущей записи.

Обновить значение полей для текущей записи с сервера можно с помощью метода `RefreshRecord`.

Рассмотрим пример формы приложения БД (рис. 21.7), в котором с помощью компонента `SimpleDataSet1` выполняется доступ к данным по технологии dbExpress.NET. Пусть требуется обеспечить возможность просмотра и редактирования записей таблицы `DEPARTMENT`, принадлежащей базе данных `EMPLOYEE.GDB` сервера InterBase.

Код модуля формы для решения поставленной задачи имеет вид:

```
unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DBXpress, Borland.Vcl.ExtCtrls, Borland.Vcl.DBCtrls, Borland.Vcl.Db,
```

```
  Borland.Vcl.Grids, Borland.Vcl.DBGrids, Borland.Vcl.DBClient,
  Borland.Vcl.SimpleDS, System.ComponentModel, Borland.Vcl.SqlExpr;

type
  TForm3 = class(TForm)
    SQLConnection1: TSQLConnection;
    SimpleDataSet1: TSimpleDataSet;
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    DBNavigator1: TDBNavigator;
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SimpleDataSet1AfterPost(DataSet: TDataSet);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form3: TForm3;

implementation

{$R *.nfm}

procedure TForm3.FormCreate(Sender: TObject);
begin
  SimpleDataSet1.Open;
end;

procedure TForm3.FormDestroy(Sender: TObject);
begin
  SimpleDataSet1.Active:=False;
end;

procedure TForm3.SimpleDataSet1AfterPost(DataSet: TDataSet);
begin
  SimpleDataSet1.ApplyUpdates(-1);
end;

end.
```

Компонент `SimpleDataSet1` работает в табличном режиме. При разработке формы подсвойству `CommandType` свойства `DataSet` задано значение `ctTable`, с помощью подсвойства `CommandText` указано имя таблицы `DEPARTMENT`.

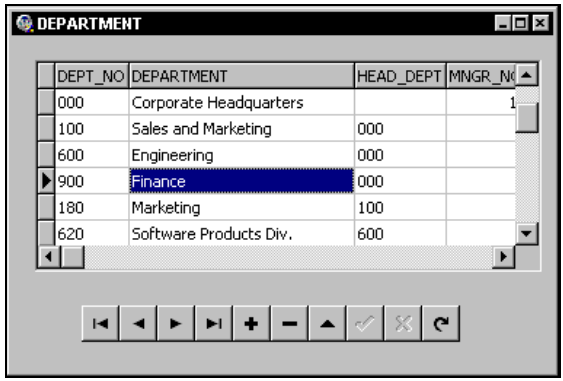


Рис. 21.7. Вид формы с заголовком `DEPARTMENT` при выполнении приложения

Для отображения записей редактируемого набора данных и навигации по нему используются компоненты `DBGrid1` и `DBNavigator1` соответственно. Фиксация внесенных при редактировании изменений и отправка данных на сервер осуществляется в обработчике события `AfterPost`, возникающего при нажатии кнопки `Post` (утвердить результат изменения записи) компонента `DBNavigator1` и при переходе в компоненте `DBGrid1` на другую строку *(см. главу 18)*.

Отладка соединения с сервером

Для получения информации о местах ошибок в SQL-запросах при отладке соединения приложения с сервером БД служит компонент `SQLMonitor`. Он осуществляет перехват сообщений между соединением и сервером баз данных и помещает их в список строк, определяемый свойством `TraceList` типа `TStrings`.

Для начала работы с компонентом нужно установить связь с соединением через свойство `SQLConnection` и активизировать его, задав свойству `Active` значение `True`.

Перед каждой записью сообщений в список строк возникает событие `OnTrace` типа

```
TTraceEvent = procedure(Sender: TObject; CBInfo: pSQLTRACEDesc;
     var LogTrace: Boolean) of object;
```

а сразу после записи в список возникает событие `OnLogTrace` типа

```
TTraceLogEvent = procedure (Sender: TObject; CBInfo: pSQLTRACEDesc)
   of object;
```

Содержимое списка можно сохранить в файле на жестком диске с помощью метода `SaveToFile`.

При задании свойству `AutoSave` типа `Boolean` значения `True` данные о проходящих командах автоматически заносятся в текстовый файл с именем, заданным значением свойства `FileName` типа `String`.

Свойство `MaxTraceCount` типа `Integer` определяет максимальное число контролируемых команд, а также управляет процессом контроля. При значении `-1` ограничения снимаются, а при значении `0` контроль не выполняется.

На число сохраненных в списке команд указывает свойство `TraceCount` типа `Integer`.

Рассмотрим следующий пример, пусть требуется выполнять оперативное отображение информации о последнем сообщении в окне многострочного редактора (компонент `memo1`). Для решения названной задачи подходит следующий код:

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject;
  CBInfo: pSQLTRACEDesc);
  var
   LogFileName: string;
begin
   with Sender as TSQLMonitor do
   begin
    memo1.Lines.Clear;
    memo1.Lines:=TraceList;
    TraceList.Clear; {очистка списка сообщений}
   end;

end;
```

Как видно из приведенного кода, мы использовали обработчик события `OnLogin`.

Предметный указатель

E

F

G

H

I

J

K

L

U

V

W

А

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ш

Э

Я