Шупрута В. В.

Delphi 2005

Учимся программировать

- Легкая и доступная форма изложения
- Все основные вопросы по работе в Borland Delphi 2005
- Большое количество примеров
- Рекомендации начинающим программистам
- Не требует начальной подготовки

САМОУЧИТЕЛЬ

NT PRESS

Оглавление

Введение

В течение последнего времени среда разработки программного обеспечения Borland Delphi 2005 была одним из наиболее ожидаемых программных продуктов. Чем вызван подобный интерес?

Причина его, прежде всего, в том, что Borland – достаточно серьезная фирма, продуктами и разработками которой на сегодня пользуются миллионы людей. Далеко не каждая фирма достигает таких результатов. Судите сами – за всю свою историю разработчиками Borland было создано достаточно большое количество инструментальных средств разработки программ для операционных систем MS DOS (Turbo/Borland Pascal, Turbo/Borland C/C++, Turbo Assembler), Windows (Delphi, C++ Builder, JBuilder) и Linux (Kylix).

Кроме того, с появлением новой версии программного продукта ожидалось и появление новых его возможностей, которые позволяли бы использовать самые последние технологии в области программирования. Следует отметить, что разработчикам Borland Delphi 2005 удалось преподнести сюрприз – ими была создана принципиально новая среда разработки программ.

Итак, Borland Delphi 2005 – это среда разработки, продукт, предназначенный для создания программ. С помощью этой среды можно достаточно быстро и эффективно создавать программы любой сложности и любого назначения. В данной книге вы найдете краткое описание возможностей этой среды разработки, а также языка программирования Delphi.

Далее будет сказано несколько слов о том, что же собой представляет среда разработки Borland Delphi 2005.

Что такое среда разработки Borland Delphi 2005

В этом небольшом введении дана самая общая характеристика системы программирования Borland Delphi 2005.

Немного истории

Среда разработки Borland Delphi 2005 – последняя на сегодня (2005 год) версия программного продукта от фирмы Borland. Версия эта возникла не случайно, она явилась следствием долгого процесса, корни которого уходят еще в 60-е годы прошлого века.

Язык программирования Паскаль (предшественник языка Delphi) был разработан профессором Н. Виртом в конце 60-х годов специально для обучения программированию студентов. В числе студентов этого выдающегося профессора Цюрихского университета были Ф. Канн и А. Хейлсберг. Канн позднее основал корпорацию Borland, а Хейлсберг долгое время был ее ведущим программистом. Под руководством Канна и Хейлсберга язык Паскаль был превращен в мощное средство разработки программ любой сложности.

Первым продуктом Borland из семейства Delphi для платформы Windows стала среда разработки Delphi 1 (а затем и Delphi 2), в основе которой был язык Object Pascal (язык Pascal, поддерживающий возможности объектно-ориентированного программирования). Среда Delphi 1 была разработана для создания программ под операционные системы Windows 3.1.

Появление новой версии Delphi 2 существенно отличало среду разработки от предыдущих продуктов. Данная версия была разработана уже под 32-разрядные операционные системы Windows 95 и Windows NT 4.

Следующие версии Delphi (3, 4, 5, 6, 7) являлись следствием постепенного развития среды разработки – улучшались существующие компоненты, добавлялись новые возможности, большое внимание уделялось программированию баз данных и программ для глобальной сети Internet. Также можно добавить и то, что к появлению седьмой версии продукта язык Object Pascal был существенно доработан и получил новое название – Delphi.

Первой версией, принципиально отличающейся от предыдущей, стала восьмая по счету версия. Принципиальное отличие ее заключается в том, что Delphi 8 была создана для разработки программ под новую перспективную платформу .NET, созданную не менее известной фирмой Microsoft.

Казалось, было бы логично, чтобы следующая версия Delphi (коей является Borland Delphi 2005) получила порядковый номер 9. Но этого не произошло. Почему?

Из основных особенностей среды разработки Borland Delphi 2005 можно отметить то, что новая среда разработки объединила в себе весь опыт программирования и создания специальных продуктов для разработки программного обеспечения. В Borland Delphi 2005 можно создавать программы для платформы Win32 (как это было раньше в версиях по седьмую включительно), а можно создавать программы и под новую перспективную платформу .NET (как в восьмой версии). Кроме того, до восьмой версии использовался язык программирования, основанный на языке Pascal (сначала Object Pascal, затем Delphi). В новой версии существует возможность разработки программ на нескольких языках (Delphi, C#, Java) – то, чего раньше не было ни в одной подобной среде разработки. Если к вышесказанному добавить и то, что редактирование программ

стало проще и удобнее за счет применения самых последних достижений в области программирования, то становится понятно, почему среда разработки Borland Delphi 2005 вышла именно под таким «автономным» названием.

Краткое описание среды разработки Borland Delphi 2005

Кратко можно отметить следующие основные особенности среды разработки:

- визуальное конструирование программ;
- использование готовых компонентов-заготовок для будущих программ;
- поддержка нескольких языков программирования;
- возможность создания программ под разные платформы;
- введение множества технологий, ускоряющих и облегчающих написание программ.

В основе идеи использования Borland Delphi 2005 при разработке программ лежит технология визуального конструирования. Что это такое?

Визуальное конструирование (Visual design) программ избавляет программиста от многих сложностей, например от рутинной разработки интерфейса (внешнего вида) будущей программы. Borland Delphi 2005 содержит все необходимые программные заготовки – кирпичики, из которых строится интерфейс программы. Программист использует прототип будущего окна программы – форму – и наполняет ее необходимыми компонентами, реализующими нужные интерфейсные свойства. При этом количество компонентов, из которых программист может «собирать» свою программу, достаточно велико. Все необходимые для создания программы компоненты объединяются в так называемую библиотеку визуальных компонентов.

Библиотека визуальных компонентов (Visual Component Library) предоставляет программисту огромное разнообразие созданных разработчиками Delphi программных заготовок, которые можно сразу использовать при написании собственных программ. При этом компоненты содержат в себе помимо программного кода и все необходимые для их работы данные, что избавляет программиста от рутинной работы по «изобретению велосипедов», – нет необходимости писать то, что уже было написано, – достаточно воспользоваться большим опытом программистов-создателей Borland Delphi 2005. Использование подобного подхода во много раз сокращает время разработки программ, а также существенно снижает вероятность случайных программных ошибок.

Представьте, как бы вам пришлось тяжко, если бы под рукой не было готовых компонентов, – время разработки программ существенно бы возросло, и еще неизвестно, осталось бы вообще у вас желание программировать или нет.

Стоит отметить, что хотя библиотека и содержит в своем названии слово «визуальных», но кроме тех компонентов, которые будут видны во время выполнения программы, она содержит также много невидимых компонентов, реализующих те или иные возможности, например стандартные диалоги, таймер, различные списки и т.д.

Поддержка нескольких языков программирования – новый этап в развитии программных продуктов подобного рода. Раньше программы создавались в различных средах, предназначенных для использования того или иного языка программирования. Кроме того, зачастую большой проблемой было столкнуться с разработкой сложных программных комплексов, элементы которых были написаны на разных языках программирования. В данной среде введена поддержка нескольких, наиболее популярных и мощных, языков программирования – Delphi, C#, Java. Кроме того, были окончательно устранены вопросы совместимости языков программирования – теперь, даже если элементы программы написаны на разных языках, никаких проблем с совместимостью не возникает.

Возможность создавать программы под *различные платформы* – также полезное достоинство Borland Delphi 2005. Казалось бы, ну появилась новая платформа .NET – ну и зачем она нам нужна? Научимся мы программировать под Windows, разве этого не достаточно?

Microsoft .NET – это новая технология разработки программного обеспечения под Windows (а в дальнейшем – и под другие операционные системы). В ее основе лежит идея универсальности программного кода, что дает возможность работы программы на любой платформе (операционной системе) при условии, что эта платформа поддерживает технологию .NET. Универсальность программного кода достигается за счет предварительного преобразования исходной программы в «нейтральный», промежуточный, код и затем с последующей трансляцией этого кода в выполняемую программу уже на этапе выполнения самой программы.

Кратко из основных особенностей .NET можно отметить следующие:

- полное межъязыковое взаимодействие. Какой бы язык программирования ни использовали при создании программ для .NET, вы не увидите никаких сложностей, поскольку все языки преобразуются в итоге к одному виду – «промежуточному» языку;
- общая среда выполнения программ, независимо от того, на каких языках они были созданы. Теперь можно с легкостью пересесть на другой компьютер и запустить свою программу. Будьте уверены – никаких казусов не произойдет, ваша программа запустится и будет работать (при условии, конечно, что на этом компьютере установлена среда .NET).

Поэтому мой вам совет – используйте эту платформу, это намного облегчит вам жизнь, если вы хотите заниматься программированием. Соответственно, довольно существенная часть книги посвящена освещению вопросов, касающихся использования именно новой технологии .NET. Конечно, о другой платформе (Win32) я не забыл – книга по сути разделена на две части, которые посвящены разработке программ для .NET и Win32 соответственно.

Еще одна важная особенность среды разработки Borland Delphi 2005 – применение множества новых технологий, *облегчающих (и ускоряющих) создание программ*. Не стоит забывать, что основными достоинствами, которые ценятся программистами (и не только ими), являются быстрота и удобство разработки программ. Быстрота достигается за счет использования готовых решений, а также за счет того, что большая часть текста (исходного кода) программы формируется автоматически, а не пишется вручную. Из удобств можно отметить грамотное оформление и возможность настройки «под себя» окна редактора исходного кода программы, а также удобную и четко структурированную справочную систему, в которой можно достаточно быстро и легко найти ответ на любой вопрос.

Несмотря на все прелести и удобства использования Borland Delphi 2005, для ее эффективного использования требуется наличие определенных навыков. Сейчас вы держите в руках книгу, которая позволит вам подружиться с Borland Delphi 2005, а также приобрести начальный опыт общения с этой средой разработки,

Что вы узнаете из этой книги

Как следует из названия, книга посвящена изучению среды разработки Borland Delphi 2005. Что же вы узнаете из этой книги?

А узнаете самое основное о среде разработки, что позволит научиться создавать собственные программы достаточно легко и быстро. Данная книга позволит приобрести начальный опыт программирования вообще и программирования на языке Delphi в частности.

Далее я приведу основные навыки, которые вы приобретете по прочтении этой книги. Итак, прочитав эту книгу, вы узнаете:

- что такое среда разработки Borland Delphi 2005;
- о планировании разработки программы;
- о том, как разрабатывать алгоритм;
- начальные сведения о языке программирования Delphi;
- основы визуального подхода к программированию;

- как пользоваться стандартными заготовками-компонентами при написании программ;
- особенности написания программ для платформы Win32;
- основы технологии .NET;
- особенности написания программ для платформы .NET.

Книга содержит достаточное количество примеров программ, исходные тексты которых можно бесплатно загрузить с сайта издательства.

Вне рамок книги остались темы использования возможностей среды разработки по программированию баз данных, создания программ, использующих возможности глобальной сети Internet и др.

Кроме того, несмотря на поддержку средой разработки нескольких языков программирования, в этой книге рассматривается только один язык программирования – Delphi. Почему именно язык Delphi? Данный язык был выбран исходя из соображений легкости его подачи начинающим (вспомните, его предок, Паскаль, был разработан именно для обучения программированию). Поэтому материала, касающегося использования языка C# или Java, вы в этой книге также не найдете.

Всей этой полезной информации нет в книге по следующей причине. Поскольку книга является достаточно кратким *учебником*, то вполне естественно, что она не содержит исчерпывающую информацию о тонкостях использования среды разработки Borland Delphi 2005. Тем более книга не является справочным пособием, где содержатся ответы на все вопросы по использованию среды разработки Borland Delphi 2005. Основной целью при создании этого учебника являлось предоставление начальной информации для той категории людей, которые собираются разрабатывать собственные программы.

Для кого предназначена книга

Книга предназначена, прежде всего, для людей, которые в силу тех или иных причин решили заняться программированием. Вполне естественно, что книгу именно для *начинающих* программистов найти достаточно сложно. Большая часть выпускаемых книг рассчитана на людей, уже имеющих какой-либо опыт в программировании.

Книга, которую вы держите в руках, написана для людей, которые подобного опыта не имеют. Она станет для них отправной точкой и позволит им сделать первый шаг навстречу сложному и интересному миру программирования.

Весь излагаемый материал написан языком, наиболее доступным и понятным этой категории людей. Кроме того, книга не содержит большого количества

сложных терминов и определений, что, безусловно, облегчает восприятие информации читателем. Если все же необходимо использовать тот или иной термин, он подается человеку в упрощенной, облегченной для понимания форме. Конечно, о точности определения (и в некотором роде правильности) в этом случае говорить достаточно сложно. Тем не менее это позволит, прежде всего, усвоить материал, а потом, по мере набирания опыта в программировании, человек сам придет к правильному пониманию того, что раньше оставалось непонятным.

Если вы уже чувствуете себя программистом со стажем, то эта книга вряд ли будет вам полезна – можете смело ее отложить: поскольку написана она как учебное пособие, то здесь нет ни подробного описания среды разработки и исчерпывающего перечня ее возможностей, ни детального описания языка программирования Delphi.

Если же вы только приступаете к изучению программирования, то книга, несомненно, станет вашим незаменимым помощником.

Обозначения, принятые в книге

Теперь скажу несколько слов об обозначениях, принятых в этой книге.

Хотя я и сказал, что в книге будет использовано минимальное количество профессиональных терминов, все же без них никуда не денешься. Кроме того, зачастую разговор идет о моментах, которым следует уделить особое внимание. Поэтому различные термины, а также отдельные важные слова, на которые надо обратить особое внимание, будут выделены *курсивом*.

Кроме того, по ходу изложения будут описываться элементы того, что мы будем наблюдать на экране, – названия пунктов меню, команд, диалоговых окон. Они для облегчения восприятия будут отмечены **полужирным начертанием**.

Также в процессе усвоения материала иногда придется выполнять последовательно несколько команд. Поначалу действия будут выполняться в виде шагов (1-й, 2-й, 3-й шаг и т.д.), затем они будут описываться в виде последовательности команд и обозначаться следующим образом – **Команда 1 ➤ Команда 2 ➤ Команда 3** и т.д.

В книге также приведено большое количество примеров исходных текстов программ. Для облегчения восприятия эти тексты (листинги) программ будут оформлены так же, как и в среде разработки Borland Delphi. Основной текст программы будет выделяться моноширинным шрифтом, ключевые (зарезервированные) слова – **полужирным**, комментарии к программам – *курсивом*.

Глава 1 Среда визуального программирования Borland Delphi 2005

Первая глава книги посвящена вопросам установки среды визуального программирования, а также краткому описанию основных ее элементов и возможностей.

Установка Borland Delphi 2005

Установка системы программирования Borland Delphi 2005 производится с компакт-диска, на котором находятся все необходимые файлы и программа установки. Программа установки запустится автоматически, когда вы вставите диск в компьютер. После этого начнется проверка вашей операционной системы на готовность к установке Borland Delphi 2005.

Для установки Borland Delphi 2005 в операционной системе должны быть установлены следующие компоненты:

- *Microsoft Internet Explorer 6.0 Service Pack 1* – последняя обновленная версия обозревателя сети Internet, необходимая в основном для отображения элементов справочной системы;
- *Microsoft .NET Framework 1.1* – платформа, обеспечивающая выполнение программ, разработанных с помощью технологии .NET;
- *Microsoft .NET Framework SDK 1.1* – набор вспомогательных программ (утилит), справочной информации и примеров, так называемый набор разработчика программного обеспечения (Software Development Kit – SDK);

➤ *Microsoft Visual J# .NET 1.1* – компонент, необходимый для разработки Internet-приложений.

Перечисленные компоненты не входят в состав Windows, поэтому их придется устанавливать отдельно. Если же какого-либо компонента нет, то программа установки автоматически завершится с предложением установить недостающие компоненты.

В остальном процесс установки Borland Delphi 2005 производится в автоматическом режиме и не должен вызвать никаких сложностей. Запускается Delphi обычным образом – командой **Пуск ➤ Все программы ➤ Borland Delphi 2005 ➤ Delphi 2005** (рис. 1.1).

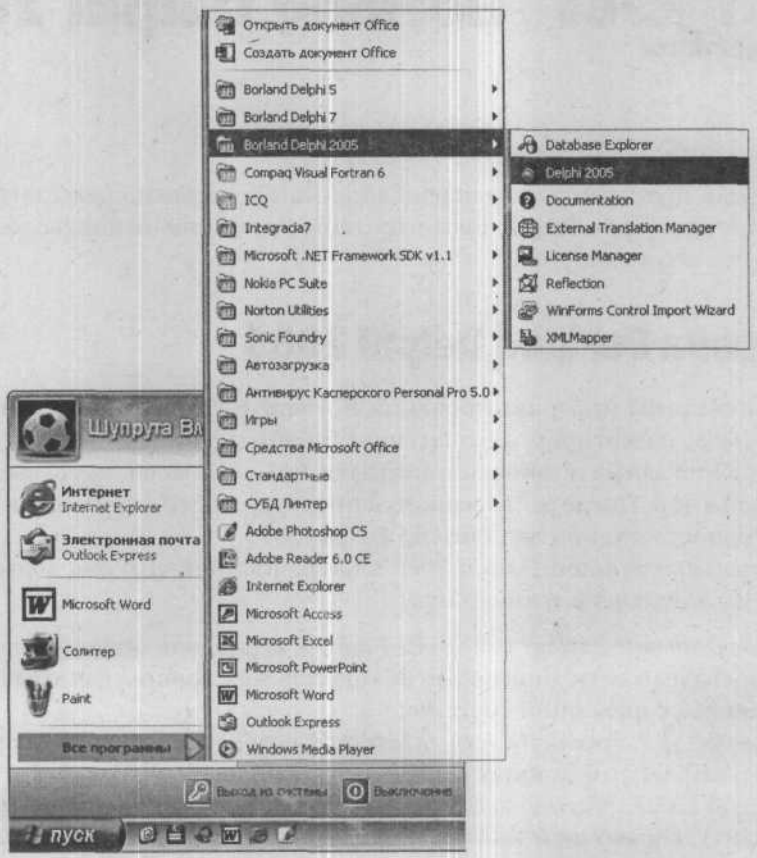

Рис. 1.1 ▼ Запуск среды разработки Borland Delphi 2005

Знакомство со средой разработки

После загрузки на экране появится главное окно среды разработки (IDE) – рис. 1.2.

Верхнюю часть окна занимает главное меню среды и панель инструментов. Центральную часть экрана занимает страница приветствия **Welcome Page**, выполненная как HTML-документ. На этой странице находятся вспомогательная информация, помогающая ознакомиться со средой разработки, а также ссылки на различные разделы справочной системы. Кроме того, если ваш компьютер подключен к глобальной сети Internet, то можно воспользоваться соответствующими ссылками для посещения сайта компании Borland и получения справки. Кроме этого, можно посетить и любой другой сайт, введя его адрес в соответствующем поле и нажав клавишу **Enter**.

В левой и правой частях экрана отображаются служебные окна для работы над проектом – окно структуры проекта **Structure**, окно инспектора

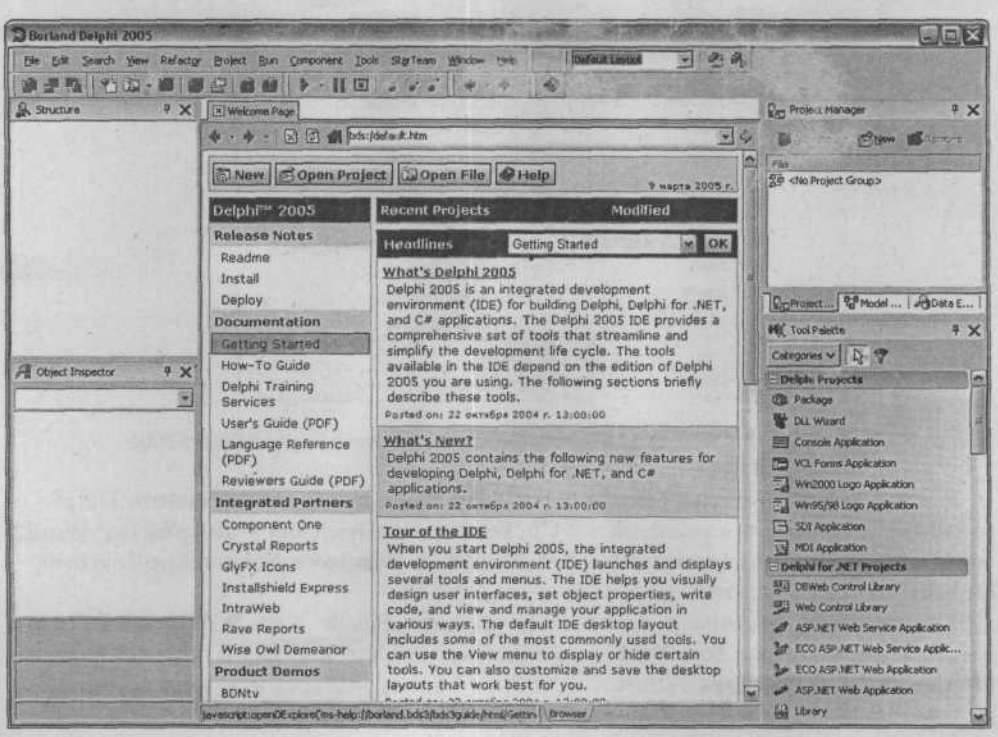

Рис. 1.2 ▾ Главное окно Borland Delphi 2005 после запуска

объектов **Object Inspector**, окно управления проектом **Project Manager**, окно просмотра модели **Model View**, окно навигатора по базам данных **Data Explorer** и, наконец, окно палитры инструментов **Tool Palette**. Назначение этих окон мы рассмотрим несколько позднее, а пока остановимся на понятии «проект».

Проект (Project) – совокупность файлов, которые используются средой разработки (точнее говоря – компилятором Borland Delphi 2005) для итоговой генерации программы. Когда мы будем создавать первый проект, то познакомимся со структурой проекта, а также со всеми составляющими его файлами.

Для того чтобы создать проект в Delphi, необходимо в меню **File** (Файл) выбрать команду **New** (Новый) и затем указать тип создаваемого проекта. Среда разработки поддерживает достаточно большое количество типов проектов, некоторые из которых приведены на рис. 1.3.

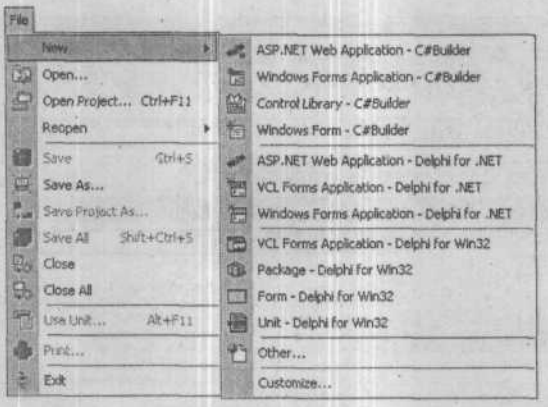

Рис. 1.3 ▼ Некоторые типы проектов, создаваемые в Borland Delphi 2005

В этой книге будут рассмотрены только язык программирования Delphi и создание двух типов проектов – **VCL Forms Applications – Delphi for Win32** (программирование для платформы Win32) и **Windows Forms Applications – Delphi for .NET** (программирование для .NET).

Попробуйте, например, выполнить команду **File ➤ New ➤ Windows Forms Application – Delphi for .NET**, что означает начало работы над проектом для операционной системы Windows для платформы .NET. После выполнения команды экран будет выглядеть, как показано на рис. 1.4.

Внимательно посмотрите на экран – содержимое окон изменилось. Далее мы подробнее рассмотрим основные окна при работе над проектом.

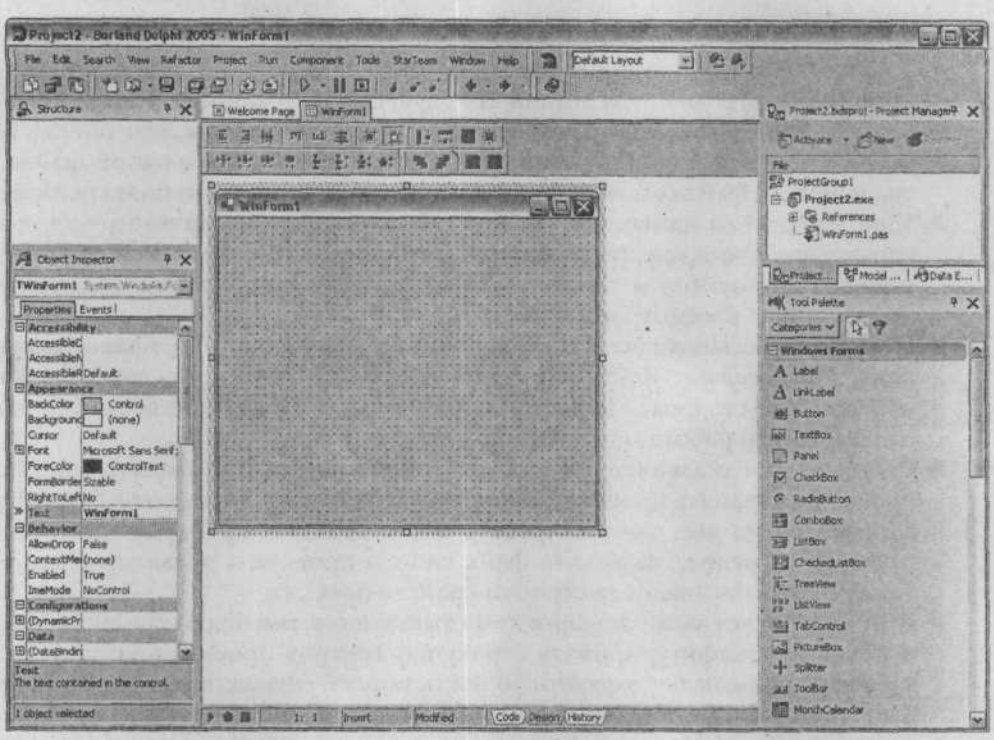

Рис. 1.4 ▾ Начало работы над проектом

Главное меню

Главное меню (Main menu) позволяет вызывать все инструменты, необходимые для работы с проектом. Рассмотрим назначение разделов меню и связанных с ними функций:

- **File** – содержит набор команд для работы с файлами, позволяет создавать новые проекты, добавлять новые файлы в проект на основе различных шаблонов, переименовывать файлы проекта, а также распечатывать их. Сюда же включена команда закрытия среды разработки;
- **Edit** – здесь, в соответствии с названием, расположены команды, предназначенные для редактирования текста, удаления и перемещения его в буфер обмена, вставки текста из буфера и отмены операций редактирования. Стоит отметить, что все эти команды работают не только с текстом, но и с компонентами в конструкторе форм – элементами управления, из которых «собирается» интерфейс вашего приложения. Кроме того, здесь же сосредоточены

команды управления положением компонентов на поверхности формы, а также фиксации компонентов – функции, позволяющей заблокировать компонент, чтобы впоследствии его случайно не изменить;

- **Search** – содержит набор команд для работы с текстом, его поиска и замены, причем и то и другое может производиться как в одном файле, так и во всех файлах проекта, либо в любом каталоге и его подкаталогах;
- **View** – под этим названием объединены команды вызова инструментов управления проектом, такие как инспектор объектов, конструктор форм, менеджер проектов и т.д. (некоторые из них, наиболее часто используемые, будут рассмотрены ниже);
- **Refactor** – здесь находятся команды, предназначенные для рефакторинга кода. *Рефакторинг* (Refactoring) – это процесс изменения кода таким образом, что его смысл (то есть выполняемые действия) остается тем же самым, но его работа и читаемость повышаются;
- **Project** – предназначен для того, чтобы добавлять и удалять модули проекта, сохранять проект в репозитории, добавлять проекты в группу и убирать их из нее, компилировать как отдельные проекты, так и все проекты в группе, загружать файл самого проекта в редактор кода, а также вызывать диалог настройки свойств проекта;
- **Run** – позволяет запускать проект на выполнение как под отладчиком, так и без него, конфигурировать строку параметров проекта при запуске, производить отладку, задавать точки останова, осуществлять пошаговое выполнение кода, просматривать значения переменных и изменять их;
- **Component** – здесь сосредоточены команды, предназначенные для установки новых компонентов и пакетов компонентов и создания новых компонентов и шаблонов компонентов;
- **Tools** – позволяет настроить свойства рабочей среды Delphi 2005 и отладчика, произвести настройку репозитория (архива проектов), добавлять и удалять дополнительные утилиты, а также команды запуска этих самых утилит;
- **StarTeam** – здесь собраны команды управления клиентом специального программного обеспечения, обеспечивающего совместную работу нескольких программистов над проектом;
- **Window** – позволяет переключаться между окнами, если вы откроете какой-либо модуль для редактирования в новом окне;
- **Help** – объединяет команды вызова справочной системы Delphi 2005 и ее настройки, а также позволяет обратиться к Web-ресурсам компании Borland для получения дополнительной информации.

Панель инструментов

Панель инструментов (Toolbar) позволяет организовать быстрый доступ к нужным вам инструментам Delphi 2005.

Вы можете настроить панель инструментов таким образом, чтобы с ней было удобно работать. Самым простым вариантом настройки является простой выбор: какие из групп «быстрых» кнопок вы хотите видеть на панели инструментов. Для этого наведите указатель мыши на панель инструментов и нажмите правую кнопку мыши или в главном меню выберите пункты **View ➤ Toolbars**. В появившемся всплывающем меню просто щелкните по имени той группы, которую хотите отобразить или скрыть (галочка слева от названия группы означает, что эта группа отображается на панели инструментов).

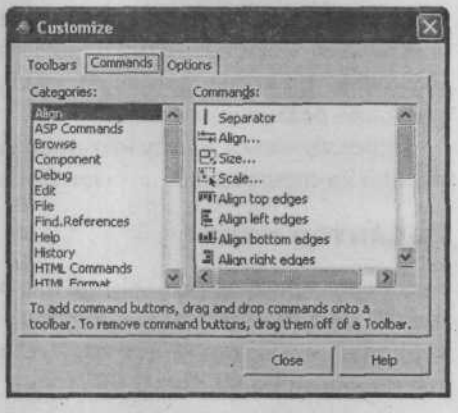

Рис. 1.5 ▾ Диалог настройки панели инструментов

Более широкие возможности по настройке панели инструментов предоставляет диалоговое окно, вызываемое через меню **View ➤ Toolbars ➤ Customize** (рис. 1.5).

На закладке **Toolbars** вы можете выбрать те группы кнопок, которые желаете видеть на панели инструментов. Как и в предыдущем случае, вам следует снять флажок слева от имени группы, чтобы скрыть ее, и поставить – чтобы показать.

Закладка **Commands**, показанная на рисунке 1.5, позволяет указать, какие кнопки нужно показывать в группе. Настройка выполняется следующим образом. Если вы хотите добавить кнопку на панели инструментов, то нажмите и удерживайте левую кнопку мыши на названии этой кнопки в списке **Commands**, перетащите ее на панель инструментов и там отпустите кнопку мыши. Если вы хотите скрыть какую-либо кнопку, то подобным образом перетащите ее в обратном направлении с панели инструментов на окно диалога настройки.

В дополнение к этому на закладке **Options** данного окна вы можете установить, показывать или нет подсказки при перемещении курсора мыши над кнопками панели инструментов (флажок **Show tooltips**) и включать или нет в подсказку комбинации «быстрых» клавиш для вызова команды, запускаемой щелчком по кнопке (флажок **Show shortcut keys on tooltips**).

Конструктор форм

Центральную часть окна теперь занимает окно *конструктора форм* (Form designer) приложения (см. рис. 1.4). *Формой* (Form) приложения на этапе разработки принято называть окно программы (во время разработки это – форма, на этапе выполнения – окно).

В окне конструктора форм вы будете формировать внешний вид своей будущей программы – изменять само окно, а также наполнять его различными элементами. Окно конструктора формы изначально находится на переднем плане и перекрывает окно редактора кода.

Переключаться между этими окнами можно нажатием клавиши **F12** (либо нажатием на специальные вкладки **Code** и **Design** в нижней части окна – см. рис. 1.4).

Редактор кода

В окне *редактора кода* (Code editor) вы и будете собственно делать то, что понимается под «написанием программ». Попробуйте активизировать окно редактора кода – и содержимое вашего экрана будет выглядеть таким образом, как это изображено на рис. 1.6.

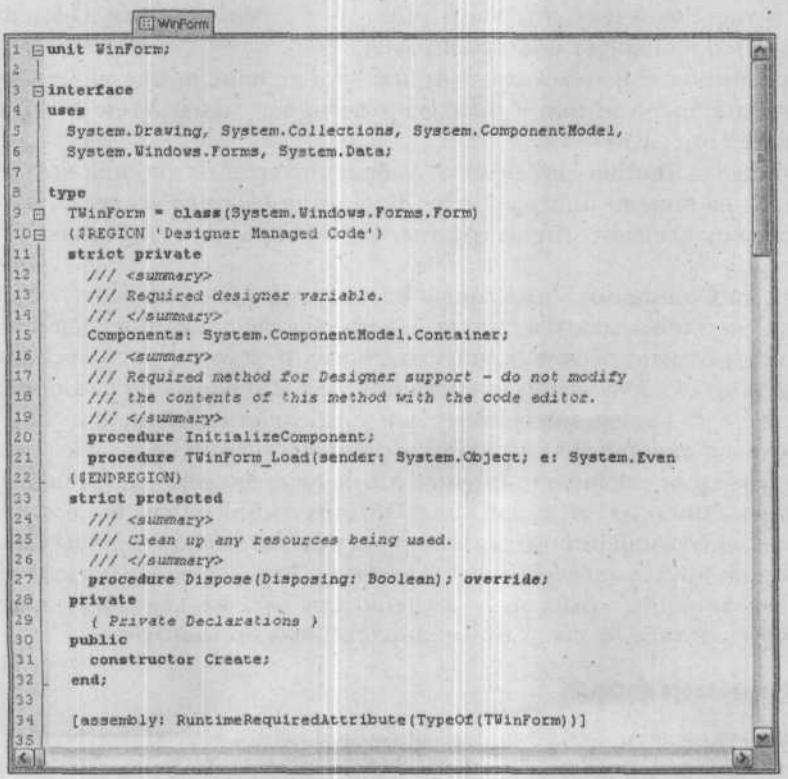

Рис. 1.6 ▼ Редактор кода

В центральной части окна теперь располагается *исходный код* вашей программы. Несмотря на то, что вы сами еще не написали ни единой строчки текста программы, это окно уже содержит код, необходимый для отображения окна программы (пока еще абсолютно пустого). Запомните, что при создании проекта у вас уже автоматически готова основа (заготовка) текста программы. Несколько подробнее с окном редактора кода мы познакомимся в конце главы, когда будем настраивать его внешний вид, а также в последующих главах, когда начнем писать свои первые программы.

Инспектор объектов

Слева от окна конструктора формы вы теперь наблюдаете измененное окно *инспектора объектов* (Object Inspector). Это окно теперь не пустое – оно заполнилось информацией выделенного объекта (в данный момент – формы). Окно **Object Inspector** имеет две вкладки (рис. 1.7) – **Properties** (Свойства) и **Events** (События).

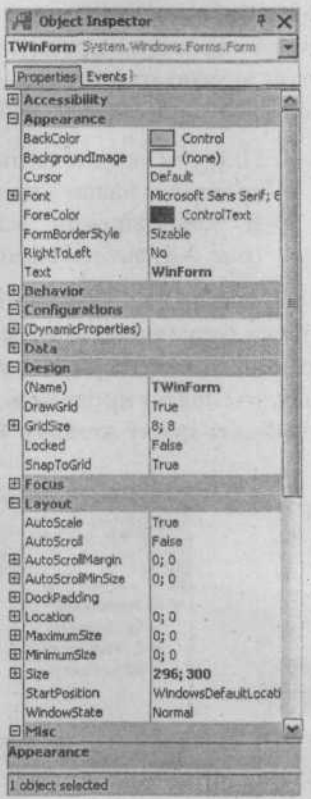

Рис. 1.7 ▾ Инспектор объектов

Первая вкладка используется для редактирования свойств объектов. *Свойство* (Property) объекта – это одна из его характеристик, которая определяет его «поведение» в программе. Объект может обладать самыми разными свойствами, которые могут объединяться в группы. Например, свойства, определяющие внешний вид объекта, объединены в группу **Appearance** (названия групп выделяются цветом). Попробуйте, например, изменить свойство `Text` – это свойство отвечает за заголовок диалогового окна. Изначально оно равно **WinForm**, измените его на любое другое и нажмите клавишу **Enter**. Вы сразу увидите, что ваша форма изменилась, – теперь она имеет тот заголовок, который вы ввели, и во время выполнения окно вашего приложения будет иметь введенный вами заголовок – значение, которое вы указали в свойстве `Text`.

Вторая вкладка окна **Object Inspector – Events** – используется для описания *событий* (Events), на которые будет реагировать выделенный объект (в данный момент – ваша форма). Подробнее

о событиях мы поговорим позднее, когда будем создавать первые проекты. Пока скажу лишь то, что каждая разработанная нами программа должна «реагировать» на те или иные события (действия пользователя) – нажатия клавиш клавиатуры, движение или нажатие кнопок мыши и т.п. Соответственно, вкладка **Events** и служит для того, чтобы определить, как будет себя вести программа.

Как я уже сказал, свойства объединяются в группы. Поначалу такая группировка может вызвать затруднения (свойств довольно много у любого объекта и для запоминания основных из них нужно некоторое время), поэтому такую группировку можно отключить. Для этого необходимо выполнить следующие действия:

1. Навести указатель мыши на окно **Object Inspector** и нажать правую кнопку мыши.
2. В появившемся вспомогательном меню (рис. 1.8) выбрать пункт **Arrange ➤ By Name**.

После этого все свойства, которые имеет объект, будут упорядочены по алфавиту (рис. 1.9). Если вы хотите вернуть окно к первоначальному виду, то следует проделать аналогичные действия и выбрать пункт **Arrange ➤ By Category**.

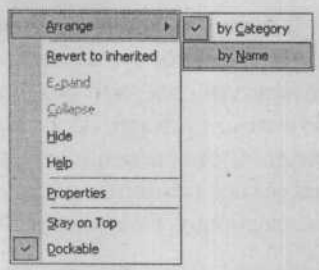

Рис. 1.8 ▼ Выбор группировки свойств объекта

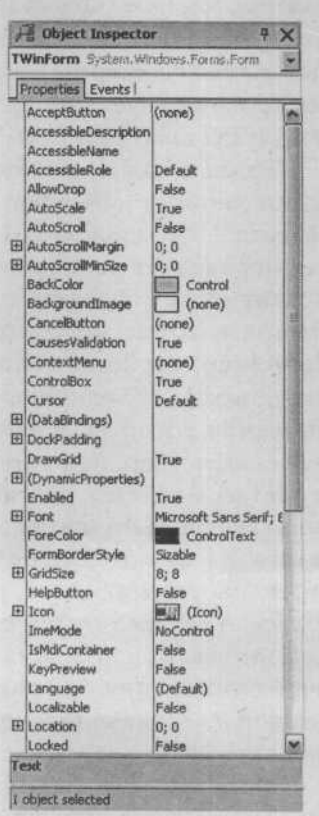

Рис. 1.9 ▼ Элементы окна **Object Inspector** теперь упорядочены по алфавиту

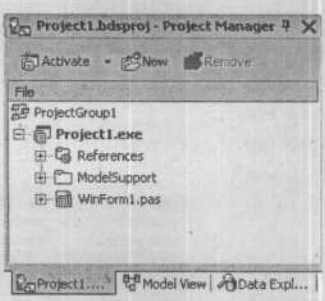

Рис. 1.10 ▼ Окно **Project Manager** теперь содержит информацию о структуре проекта

Менеджер проекта

В окне *менеджера проекта* (Project Manager), которое показано на рис. 1.10, теперь отображается структура приложения (проекта, над которым вы сейчас работаете). С помощью указателя мыши можно раскрывать содержимое списка, нажав на изображение знака «+» слева от элементов списка.

Зачем может понадобиться подобное окно? В этом окне содержится общая информация о проекте, информация об используемых внешних модулях (библиотеках), а также обо всех файлах проекта. Попробуйте выделить пункт **Project1.exe** в данном окне и нажать правую кнопку мыши. Перед вами появится контекстное меню, показанное на рис. 1.11.

В этом меню можно, например, сохранить проект (**Save**, **Save as...**), переименовать его (**Rename**), посмотреть исходный код (**View Source**), преобразовать (скомпилировать) проект в исполняемый файл (**Compile**) и т.д.

Также с помощью этого окна, используя кнопки **New** (Новый элемент) и **Remove** (Удалить элемент), вы сможете добавлять в проект или удалять из проекта модули и формы.

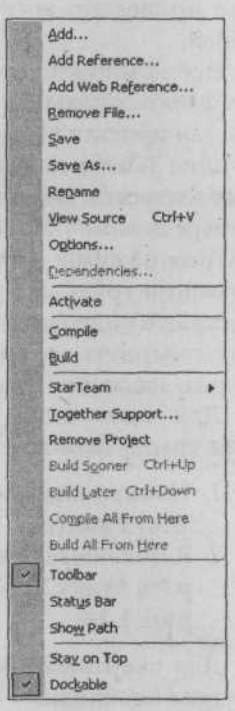

Рис. 1.11 ▾ Меню для работы с проектом

Палитра компонентов

Палитра компонентов (Tool Palette) – это один из наиболее часто используемых инструментов Delphi (рис. 1.12). Она состоит из большого числа групп, в которых располагаются компоненты.

Компонент (Component) – это элемент пользовательского интерфейса, который может быть перенесен на форму. Это могут быть кнопки, метки, поля для ввода всевозможных данных, выпадающие списки – в общем все то, что вы обычно видите на окнах в операционой системе Windows (такие компоненты называют *визуальными* (Visual)). Кроме того, это могут быть также и *невидимые* компоненты (чаще употребляют слово *невизуальные* (Non-visual)), то есть те компоненты, которые не отображаются в момент выполнения программы, но выполняют различные функции. Типичный пример такого компонента – таймер (`Timer`). Во время создания первых проектов мы постепенно будем знакомиться

с компонентами, а более подробное их описание приведено во второй и третьей частях книги.

Все компоненты объединяются в группы по функциональному назначению. После создания проекта раскрыт список компонентов группы **Windows Forms**, содержащий основные элементы диалоговых окон Windows. Посмотреть компоненты других групп можно нажатием на символ «+», находящийся слева от названия группы. Кроме того, можно легко раскрыть содержимое всех групп или наоборот – свернуть все списки, чтобы были видны только названия групп.

Для того чтобы раскрыть содержимое всех групп, необходимо:

1. Нажать правую кнопку мыши в окне **Tool Palette**.
2. В появившемся контекстном меню выбрать пункт **Expand All** (Раскрыть все) – рис. 1.13.

Для свертывания списка компонентов в группы необходимо выполнить аналогичные действия и в контекстном меню выбрать пункт **Collapse All** (Свернуть все).

Рис. 1.12 ▼ Окно палитры компонентов

Как видите, количество компонентов достаточно велико. Вполне естественно, что не все эти компоненты (группы компонентов) будут вам полезны, поэтому можно отключить отображение ненужных групп компонентов. Сделать это можно с помощью уже известного нам контекстного меню, выбрав пункт **Delete <имя_категории> Category** (Удалить <имя_категории> группу компонентов). Если же по каким-либо причинам вы удалили группу, которая вам снова понадобилась (или просто удалили слишком много групп), то всегда можно восстановить исходный набор компонентов, выбрав в контекстном меню пункт **Reset Palette** (Восстановить палитру компонентов). На начальном этапе вы можете поэкспериментировать с отображением элементов этого окна, но особенно увлекаться этим делом не стоит.

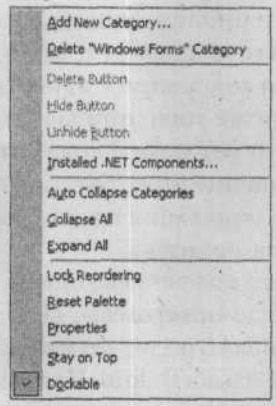

Рис. 1.13 ▾ Контекстное меню окна **Tool Palette**

Структура проекта

Окно *структуры проекта* после создания проекта отображается в левой верхней части экрана. Это окно содержит информацию о структуре исходного кода программы (именно поэтому оно не содержит информации, если активно окно дизайнера формы). Для того чтобы посмотреть данную информацию, следует переключиться в окно редактора кода (с помощью клавиши **F12** или вкладки **Code** в нижней части экрана). После активизации окна редактора кода окно **Structure** заполняется информацией (рис. 1.14).

В данный момент, скорее всего, вам мало понятно содержимое этого окна. Поэтому заострять внимание на этом не стоит – как только мы познакомимся со структурой проекта (в данном случае – проекта для платформы .NET),

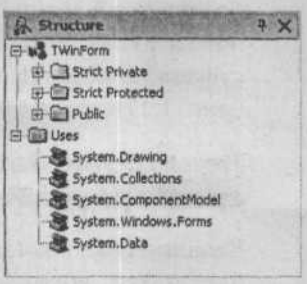

Рис. 1.14 ▾ Окно **Structure**

особенностями языка Delphi и напишем свои первые программы, то все станет на свои места. Пока отметьте для себя то, что в любой программе существуют определенные разделы и содержимое этого окна как раз и служит для отображения этих разделов. Кроме того, при выполнении двойного щелчка по какому-либо из разделов окно редактора кода автоматически переместит вас в этот раздел. Такое перемещение может быть удобно в случае, если текст программы достаточно велик и навигация по нему обычным способом (с помощью полос прокрутки) весьма неудобна.

Данное окно также может быть полезно для оперативного контроля синтаксических ошибок в исходном коде программы. При вводе текста программы среда разработки автоматически контролирует введенные данные на правильность с точки зрения синтаксиса языка Delphi. Поэтому, если вы набрали текст программы, а в окне **Structure** вдруг появилась вкладка **Errors**, значит, в тексте программы при наборе допущены ошибки (рис. 1.15).

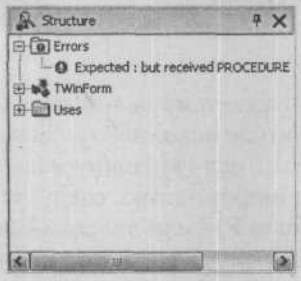

Рис. 1.15 ▼ Окно **Structure** также может содержать нформацию и об ошибках

Попробуйте в окне редактора кода удалить какую-либо строку или часть строки. Например, удалите служебное слово `interface`, которое находится сразу за строкой **`unit`** `WinForm;`. В результате в окне **Structure** появится надпись (рис. 1.16), свидетельствующая о том, что в структуре программы допущена ошибка (в данном случае – отсутствует раздел `interface`).

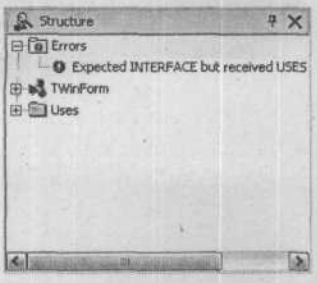

Рис. 1.16 ▼ Выдача сообщения об ошибке – отсутствует раздел `interface`

Если же по каким-то причинам вы не видите какое-либо из вышеперечисленных окон, то можете в главном меню среды разработки найти пункт **View** и затем выбрать соответствующую команду и отобразить нужное окно (рис. 1.17).

Некоторые настройки среды разработки

Казалось бы – зачем рассматривать вопросы настройки, когда мы даже не начали писать программы, да и вообще лишь поверхностно ознакомились со средой разработки? Дело в

следующем. Данная глава посвящена прежде всего знакомству с Borland Delphi 2005. При этом, пока мы знакомимся со средой, я не рассматриваю всех ее особенностей, а только лишь указываю на то основное, с чем придется иметь дело в дальнейшем. Конечно, по ходу рассмотрения материала я буду так или иначе затрагивать вопросы настройки среды, но для читателя будет удобнее, если вся информация по настройке среды будет собрана в одном месте.

В рамках знакомства рассмотрим и возможности настройки некоторых элементов среды разработки. Необходимо заметить, что мы будем настраивать только пользовательский интерфейс среды, не касаясь конкретно вопросов программирования. На данном этапе назначение некоторых элементов, которые мы можем настраивать, может показаться не совсем понятным. В подобных случаях можно смело пропускать эти пункты главы – в процессе освоения со средой всегда можно вернуться в это место и выполнить те настройки, назначение которых раньше оставалось непонятным.

Итак, начнем изучать основные вопросы настройки среды. Все, что только можно настраивать в Boland Delphi 2005, можно найти в пункте **Tools** (Сервис) главного меню. Данное меню содержит несколько подпунктов (рис. 1.18), но нам непосредственно для настройки необходимо выбрать пункт **Options** (Параметры).

После выбора пункта **Options** на экране появится окно, внешний вид которого показан на рис. 1.19.

Далее будут рассмотрены некоторые вопросы настройки самой среды (**Environment Options**), а также вопросы настройки внешнего вида окна редактора исходного кода (**Editor Options**).

Рис. 1.17 ▾ Меню **View** позволит отобразить нужное окно

Рис. 1.18 ▾ Пункты меню **Tools**

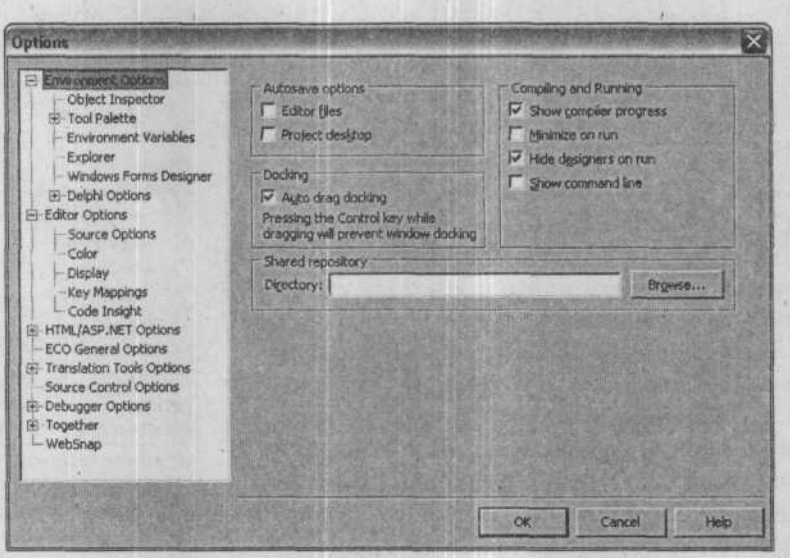

Рис. 1.19 ▼ Окно настройки параметров среды Borland Delphi 2005

Общие настройки среды

В группу общих настроек среды входят настройка рассмотренных нами ранее окон **Object Inspector**, **Tool Palette**, **Structure**, а также задание параметров для окна дизайнера формы при программировании для платформ Win32 и .NET. Более подробно основные параметры настройки приведены в табл. 1.1.

Таблица 1.1 ▼ Некоторые параметры настройки пункта **Environment Options**

Пункт настройки	Настраиваемые элементы	Комментарий
Object Inspector	**Speed Settings**	Быстрое задание цветовой схемы и настроекокна путем выбора из списка
	Colors	Задание цвета отдельных элементов окна **Object Inspector**
	Options	Настройки окна. Устанавливая или сбрасывая переключатели, можно задавать различные свойства окна, например отображение доступных только для чтения свойств – **Show read only properties**, *отображение панели* состояния в нижней части окна – **Show Status Bar** – и т.п.

Таблица 1.1 ▼ Некоторые параметры настройки пункта **Environment Options** (окончание)

Пункт настройки	Настраиваемые элементы	Комментарий
Tool Palette	**Options**	Группа свойств, позволяющих задавать размер кнопок палитры, а также признак необходимости подписывать названия к кнопкам, автоматического раскрытия списка компонентов и т.п.
	Colors	Установка цветовой палитры для окна **Tool Palette**
Explorer (настройки окна **Structure**)	**Explorer Options**	Группа свойств, задающих основные свойства окна
	Explorer Sorting	Выбор способа сортировки элементов окна – по алфавиту или же по положению в исходном коде программы
	Explorer Categories	Перечень отображаемых элементов в окне **Structure**
Windows Form Designer	**Grid Options**	Группа свойств, определяющих свойство дизайнера формы Windows Form. В этой группе определяется необходимость отображения сетки на форме, привязки перемещения объектов к сетке, а также и размер сетки
Delphi Options ➤ VCL Designer		Группа свойств, определяющих свойство дизайнера формы VCL. В этой группе определяется необходимость отображения сетки на форме, привязки перемещения объектов к сетке, а также и размер сетки. Кроме того, также может указываться необходимость отображения подсказок и названий компонентов на форме

Настройка внешнего вида окна редактора исходного кода

В группу настроек внешнего вида окна редактора исходного кода входит настройка цветового оформления, шрифта и особенностей форматирования символов. На рис. 1.20 приведен вид окна настроек при задании цветовой палитры редактора исходного кода – вкладка **Editor Options ➤ Color**.

Настраивать элементы цветовой палитры достаточно просто. В основной части окна приведен типовой фрагмент исходного кода. Выбирая в этой части окна тот или иной элемент, можно задавать цвет для выбранного элемента (Foreground Color), а также цвет фона (Background Color) для него. При выборе элемента его тип отображается в поле **Element**. В главе, посвященной знакомству с языком Delphi, вы узнаете, из каких элементов состоит программа, а

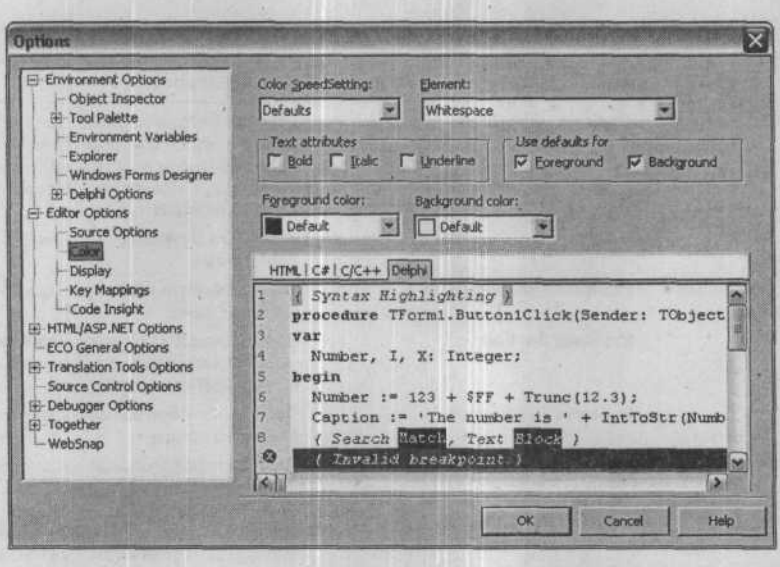

Рис. 1.20 ▼ Настройка цветовой палитры редактора исходного кода

пока запомните, что различные элементы программы для облегчения восприятия исходного кода окрашиваются в различные цвета.

Для выводимого текста можно применить форматирование – задать комбинацию параметров, определяющую начертание шрифта. Сделать это можно, использовав соответствующие переключатели в группе параметров **Text Attributes**.

Кроме этого, для быстрой смены цветовой схемы редактора кода существует несколько стандартных заготовок. Выбрать одну из заготовок можно с помощью пункта **Color SpeedSetting** (Быстрое задание настроек цвета).

Для задания остальных настроек можно использовать вкладку **Editor Options ➤ Display**. После активизации вкладки внешний вид окна настроек несколько изменится (рис. 1.21).

В группе параметров (Margin and gutter) можно определить:

- ➤ будет ли отображаться вертикальная полоса-ограничитель в правой части экрана (Visible Right Margin), а также номер символа, за которым будет вычерчиваться полоса;
- ➤ будет ли отступ в левой части экрана (Visible Gutter), а также величину (ширину этого отступа в пикселях);
- ➤ надо ли показывать номера строк (Show Line Numbers).

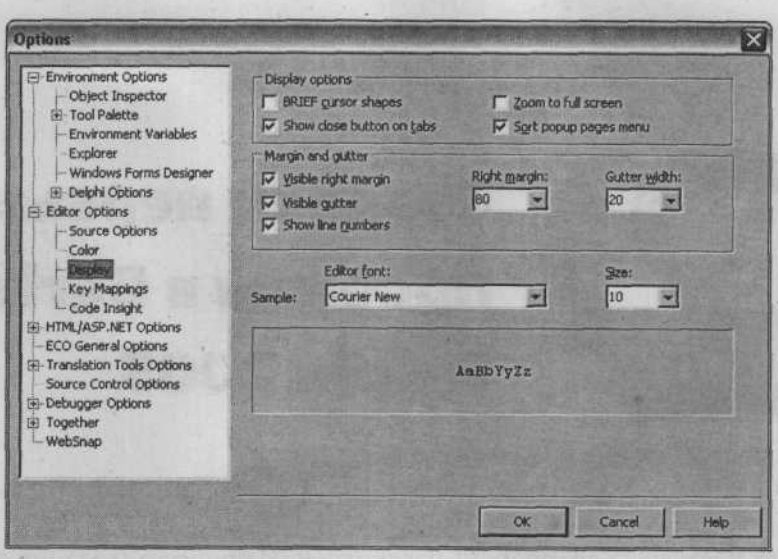

Рис. 1.21 ▾ Настройка шрифта и особенностей форматирования символов

Кроме этого в поле **Editor Font** (Шрифт редактора) можно задавать шрифт, который будет использован для вывода исходного кода. В поле **Size** (Размер) можно задавать размер выводимого шрифта в пунктах (1 пункт равен примерно 0,353 мм). Задавая тип шрифта и его размер, ниже можно сразу наблюдать, как это будет выглядеть на экране.

На этом мы заканчиваем первое знакомство со средой разработки Borland Delphi 2005. В последующих главах и частях книги основное внимание будет уделено созданию программ, а также рассмотрению особенностей использования компонентов среды и ее основных графических возможностей.

Глава 2 Создаем первые проекты в Borland Delphi 2005

Глава посвящена подробному рассмотрению процесса создания проектов в среде разработки Borland Delphi 2005. Далее будет рассмотрено два основных типа проекта с использованием языка программирования Delphi – проекты для платформ .NET и Win32. В качестве примера мы создадим учебную программу для нахождения корней квадратного уравнения (для .NET и Win32 соответственно).

Перед тем как писать программы

В этом небольшом разделе будут рассмотрены базовые понятия программирования, которые будут нам необходимы для дальнейшей работы в Borland Delphi 2005. Само по себе написание программ сводится в итоге к составлению определенных команд, которые должен выполнить компьютер.

Выражение «написать программу» на самом деле отражает только один из этапов создания компьютерной программы. Под этим выражением очень часто ошибочно понимается именно написание команд (инструкций) в текстовом редакторе (или же на бумаге). На самом деле процесс написания программы – это процесс, состоящий из нескольких этапов. Почему нескольких, спросите вы? Во-первых, прежде чем что-то «писать» или «вводить в компьютер», надо сначала определиться с тем, что же мы будем писать. Кроме того, процесс написания программы не заканчивается написанием команд (исходного кода программы) – после этого еще необходимо будет проверить ее работу. Поэтому

во избежание дальнейших осложнений считаю необходимым, насколько это возможно, кратко пояснить, что представляет собой процесс написания программы.

Определяем этапы разработки программ

Начнем с определения того, что же такое программирование. Итак, *программирование* – процесс создания (разработки) программы, который может быть представлен последовательностью следующих шагов:

1. Определение требований к программе.
2. Разработка (выбор) алгоритма решения поставленной задачи.
3. Написание команд (инструкций).
4. Отладка.
5. Тестирование.

В целом вот такой вот процесс. Сложно? На первый взгляд – может быть. Но не стоит раньше времени бить тревогу. Разберемся теперь более подробно, что же происходит на каждом из этих этапов.

Определение требований к программе – один из самых важных этапов, на котором подробно описывается исходная информация и формулируются требования к результатам. Кроме того, на этом этапе также может описываться поведение программы в особых случаях.

Поясню вышесказанное на примере. Давайте попробуем сформулировать требования к программе, которая будет решать квадратные уравнения. Требования в данном случае могут быть сформулированы следующим образом (в качестве исходных данных программа использует коэффициенты квадратного уравнения (a, b, c)):

- исходные данные (то есть коэффициенты) вводятся с клавиатуры в режиме диалога во время работы программы;
- программа должна выводить на экран значение корней уравнения;
- при отсутствии корней программа должна выводить соответствующее сообщение.

Для составления подобной программы этих четырех требований вполне достаточно. К указанным выше требованиям можно также дополнить и то, что к программам, работающим в среде Windows, могут предъявляться дополнительные требования по внешнему виду диалоговых окон. Подводя итоги, можно сказать, что на данном этапе нужно, по возможности, точно определить,

какие исходные данные необходимы программе, что она должна делать и какой должен быть результат. Переходим к следующему этапу – разработке алгоритма.

На этапе *разработки алгоритма* мы должны определить последовательность действий, которые необходимо выполнить для достижения результата, то есть *алгоритм* решения задачи. Под выбором алгоритма я подразумеваю то, что для некоторых задач существует обычно несколько способов решения, и если они известны, то можно выбрать наиболее подходящий. Результатом составления алгоритма является его словесное описание или графическое представление (чаще используют термин *блок-схема*).

После того как определены требования к программе и составлена блок-схема, можно переходить к этапу *написания команд* или собственно *кодированию*. Как видите, сразу «писать программы» не имеет смысла. Конечно, вы вполне можете «прокрутить» все требования и блок-схемы с возможными вариантами решения у себя в голове, но не стоит забывать о том, что рано или поздно вы начнете писать программы гораздо более сложные, чем те, что рассчитывают корни квадратного уравнения. Поэтому если вы привыкнете к подобному методу работы, то в дальнейшем не сможете писать сложные программы либо они будут соответствующего качества.

Итак, команды написаны. Что дальше? А дальше начинается *процесс отладки*. Под отладкой называется процесс поиска и устранения ошибок в программе. Скажу сразу – ошибки будут. Вполне вероятно, что поначалу ошибок будет много. Почему они возникают? Ошибки в написании команд могут возникать по самым разным причинам – от невнимательности при написании до неверно определенных способов получения результата. Соответственно, ошибки бывают *синтаксические* (ошибки в тексте) и *алгоритмические*, или *логические*. С первым типом ошибок все более-менее просто – обычно их достаточно легко заметить самому, да и современные системы программирования (к коим относится и Borland Delphi 2005) в этом деле очень хорошие помощники. А вот с алгоритмическими ошибками все гораздо сложнее – если неверно определен путь (способ) получения результата, то правильный результат вы не получите. Результатом этапа отладки будет *правильное* функционирование программы при нескольких вариантах исходных данных. Казалось бы, на этом можно было бы и завершить разработку программы. Все было бы так, если бы вы пользовались своими программами дома и никому их не показывали. Но если вы создаете что-то такое, что будут использовать другие, то должны позаботиться и о тех, для кого вы свои программы пишете. Поэтому немаловажным будет этап тестирования вашей программы.

Тестирование – заключительный этап. На этом этапе следует проверять свою программу на как можно большем количестве наборов входных данных, в том числе и на заведомо неверных. Кроме того, вполне вероятно, что некоторые моменты, которые, возможно, и не влияют на правильность результата, не были учтены или были учтены неверно. Подобное замечание в основном относится к элементам графического интерфейса программы. Довольно часто случается так, что все силы тратятся на получение правильного результата, а вот про то, что пользователь может «нажать что-нибудь не то» или «ввести что-нибудь не то», часто забывается. Естественно, что такие моменты, создающие эффект «недоделанной» программы, никак ее не красят.

Несколько слов о составлении алгоритма

На первом этапе создания программы программист должен определить последовательность действий, которые необходимо выполнить, чтобы решить поставленную задачу. В качестве примера мы попробовали сформулировать требования к программе, которая должна решать квадратные уравнения. Теперь попробуем выполнить следующий этап – расписать всю последовательность действий, которая позволит нам получить значения корней уравнения, другими словами, – займемся разработкой алгоритма.

Итак, *исходными данными* для решения будут коэффициенты уравнения, *результатом* же – значения корней либо сообщение о том, что уравнение их не имеет.

Метод решения квадратного уравнения известен, то есть существуют формулы, по которым мы можем их вычислить. Последовательность шагов решения поставленной задачи также очевидна. Сначала необходимо найти (вычислить) значение дискриминанта. Затем, если полученное значение больше или равно нулю, вычислить по известным формулам значения корней. Задавая последовательность действий, которые нам необходимо выполнить, мы постепенно подходим к понятию «алгоритм».

Алгоритм – это точное предписание, определяющее процесс перехода от исходных данных к результату.

Далее приведу пример составленного алгоритма решения квадратного уравнения.

Исходные данные для алгоритма – коэффициенты уравнения: a – при второй степени неизвестного, b – при первой, c – при нулевой степени.

Результат – значения корней уравнения.

Необходимые действия:

1. Вычисляем дискриминант по формуле $d = b^2 - 4ac$.
2. Если значение дискриминанта отрицательно, то уравнение корней не имеет. В противном случае корни рассчитываются по формулам:

$$x_1 = \frac{-b - \sqrt{d}}{2a}; \quad x_2 = \frac{-b + \sqrt{d}}{2a}.$$

Алгоритм решения квадратного уравнения может быть представлен и в графическом виде, то есть в виде блок-схемы. В блок-схемах для обозначения различных элементов программы используются определенные стандартные фигуры. Некоторые из этих символов приведены на рис. 2.1.

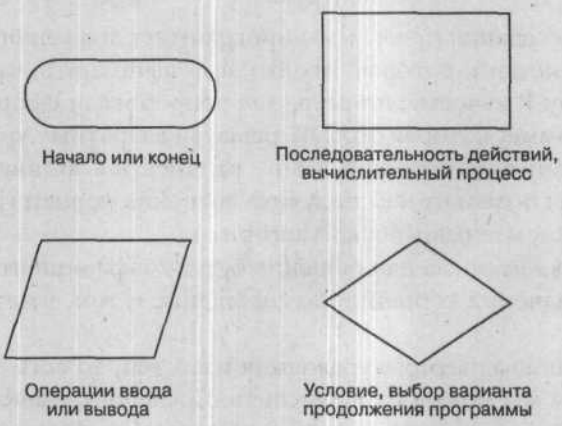

Рис. 2.1 ▼ Основные элементы, используемые при составлении блок-схемы

Основными элементами блок-схемы являются обозначения начала и конца алгоритма, обозначения ввода исходных данных и вывода результата, а также обозначение выбора, последовательности команд, переходов и т.п.

В случае использования блок-схемы ход решения задачи становится более понятным и наглядным. На рис. 2.2 представлена блок-схема алгоритма, реализующего решение квадратного уравнения. После разработки такой схемы можно уже переходить к написанию программы – последовательности команд на языке программирования, соответствующему разработанному алгоритму.

Теперь, когда есть алгоритм, можно писать программу. Если бы мы уже знали язык программирования Delphi, то, несомненно, сделали бы это.

Поэтому теперь нам предстоит немного познакомиться с языком Delphi и его возможностями. Конечно, мы не будем изучать *все* его особенности – объем данной книги не позволяет сделать этого. Да этого на самом деле и не требуется – для начала нам будет достаточно лишь ознакомиться с этим языком программирования.

В первых проектах для платформ .NET и Win32, создание которых «от и до» описано далее, вы познакомитесь с тем, как пишутся программы, и отметите для себя некоторые особенности использования среды разработки. Более подробно с языком программирования Delphi вы познакомитесь в главе 3.

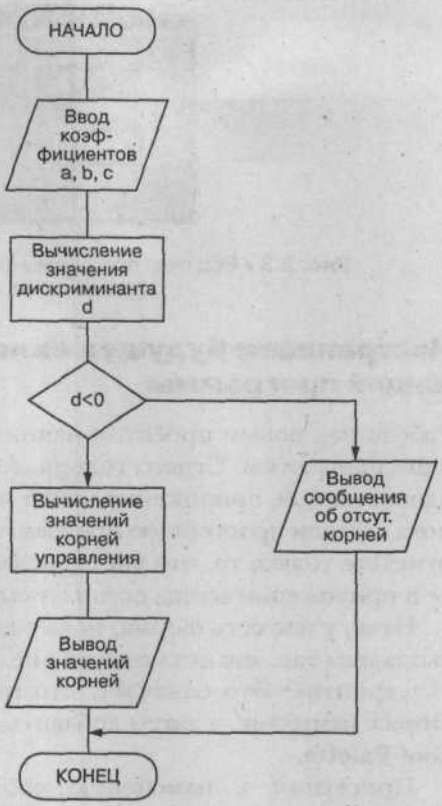

Рис. 2.2 ▾ Пример алгоритма программы, вычисляющей значения корней квадратного уравнения

Создаем первый проект для Microsoft .NET Framework

В этой главе мы постараемся максимально отойти от теории и попробуем создать свое первое приложение (программу). В первом учебном проекте мы будем разрабатывать программу, которая рассчитывает корни квадратного уравнения. Вид окна программы (после нажатия на кнопку **Расчет**), которое мы будем разрабатывать, показан на рис. 2.3. Чтобы начать работу над проектом, запустите Borland Delphi 2005 и выполните команду **File ➤ New ➤ Windows Forms Application – Delphi for .NET**.

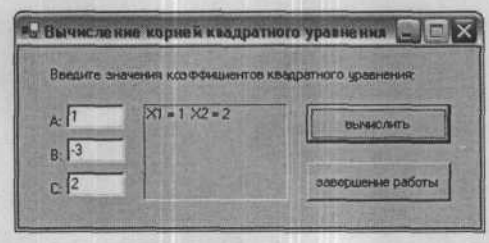

Рис. 2.3 ▼ Вид окна программы вычисления корней квадратного уравнения (после нажатия кнопки **Вычислить**)

Настраиваем будущее окно нашей программы

Работа над новым проектом начинается с создания *формы* (Form) – главного окна программы. Строго говоря, форма в приложении не обязательно будет единственной, приложение может иметь много форм. Но в нашем примере мы пока создаем простейшую программу, которая имеет всего одно окно. Для себя отметьте только то, что вне зависимости от общего количества форм в проекте в приложении всегда есть *главная* (Mainform) форма.

Итак, у нас есть форма, пока еще совершенно пустая. Для того чтобы она выглядела так, как показано на рис. 2.3, ее необходимо *настроить*. Что значит «настроить»? Это означает, что необходимо изменить ее свойства в окне **Object Inspector**, а затем добавить на нее необходимые компоненты из окна **Tool Palette**.

Приступим к изменению свойств формы – активизируйте вкладку **Properties** окна **Object Inspector**. Не пугайтесь большого количества ее свойств, нам для настройки формы будет достаточно использования лишь некоторых из них.

Попробуйте самостоятельно изменить значение свойства `Text` – щелкните кнопкой мыши по строке **Text** и введите текст «Вычисление корней квадратного уравнения», затем нажмите клавишу **Enter**. Заметьте, что те значения, которые изменил программист во время настройки формы, отображаются полужирным шрифтом.

Некоторые свойства могут быть сложными (то есть объединять в себе несколько свойств). Это, например, свойство `Size`, отвечающее за размер формы. Данное свойство объединяет в себе два свойства `Size.Width` и `Size.Height`, соответственно определяющие размеры формы по горизонтали и вертикали. Все составные свойства характеризуются наличием значка

раскрывающегося списка слева от их названия. Нажимая на этот значок, вы можете просмотреть весь список уточняющих свойств.

При выборе некоторых свойств в поле для ввода появляется кнопка с тремя точками. Это значит, что значение этого свойства нужно задать в отдельном диалоговом окне, которое появится при нажатии этой кнопки. Попробуйте выбрать свойство `Font`, позволяющее настроить параметры шрифта формы. Вы видите, что оно также содержит дополнительные свойства, которые можно посмотреть с помощью нажатия значка «+». Значения этих же свойств можно установить и другим способом – нажатием кнопки с тремя точками (рис. 2.4).

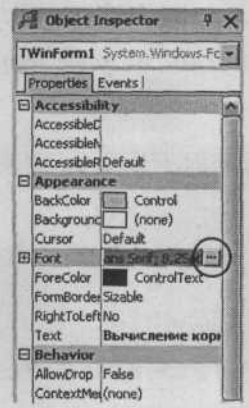

Рис. 2.4 ▾ Альтернативный способ задания свойств объекта

Далее приведена таблица, описывающая свойства формы, которые нам необходимо изменить в первом проекте (чтобы окно формы выглядело так, как на рис. 2.5), а также краткую расшифровку назначения этих свойств. Сразу отмечу, что в табл. 2.1 показаны лишь те свойства, которые *мы изменили*, значения же остальных свойств не приводятся.

Таблица 2.1 ▾ Свойства формы программы вычисления квадратных корней уравнения

Свойство	Значение	Комментарий
`Text`	Вычисление корней квадратного уравнения	Заголовок формы
`Size.Width`	392	Размер формы в пикселях по горизонтали
`Size.Height`	184	Размер формы в пикселях по вертикали

Таблица 2.1 ▼ Свойства формы программы вычисления квадратных корней уравнения (окончание)

Свойство	Значение	Комментарий
`Font.Name`	Microsoft Sans Serif	Наименование используемого шрифта (для компонентов, которые будут помещены на форму)
`Font.Size`	8	Размер используемого шрифта
`FormBorderStyle`	FixedSingle	Свойство определяет доступность изменения размеров окна программы во время выполнения. Значение свойства `FixedSingle` запрещает такое изменение
`StartPosition`	CenterScreen	Задает место появления окна программы на экране: `CenterScreen` – в центре экрана
`MaximizeBox`	False	Доступность кнопки **Развернуть** в правом верхнем углу программы. `False` – кнопка недоступна

Если вы все сделаете правильно, то ваша форма будет выглядеть так, как показано на рис. 2.5.

Рис. 2.5 ▼ Вид формы после задания основных свойств

Придаем программе внешний вид с помощью стандартных компонентов

Вы уже настроили основу для своей будущей программы – форму, однако создаваемое нами окно пока что мало напоминает то, к чему мы в итоге должны прийти. Чтобы на нашем окне появились кнопки, места для ввода данных пользователем и отображения результата, необходимо использовать компоненты. *Компонентами* (Components) называются все поля для ввода, кнопки, а также другие элементы пользовательского интерфейса. На данном этапе мы познакомимся только с теми компонентами, которые необходимы для создания нашей программы, а более подробно основные компоненты описаны в главах 5 и 7.

Вернемся к нашей программе. Нам необходимо рассчитать корни квадратного уравнения. Соответственно, программа должна получить от пользователя исходные данные о коэффициентах уравнения. Эти данные можно вводить,

например, с помощью полей для редактирования. Для этого к форме необходимо добавить соответствующий компонент – TextBox. Для добавления этого компонента на форму выполним следующие шаги:

1. Активизируйте окно **Tool Palette**, если оно недоступно.
2. В окне **Tool Palette** раскройте вкладку **Windows Forms**, затем щелкните по значку компонента **TextBox** (рис. 2.6).
3. Переместите указатель мыши в то место формы, где бы вы хотели поместить компонент.
4. Щелкните мышью еще раз.

Рис. 2.6 ▾ Добавление компонента TextBox на форму

В результате на форме появится компонент TextBox – поле для редактирования (рис. 2.7).

Каждому компоненту, помещаемому на форму, присваивается имя (имя есть у всех компонентов, оно содержится в свойстве Name). Это необходимо для того, чтобы программист мог получить доступ к его свойствам. По умолчанию наш компонент называется в программе TextBox1. Если мы добавим еще такие компоненты, то они будут названы соответственно TextBox2, TextBox3 и т.д. Совсем не обязательно пользоваться такими именами – вы можете задавать любое имя компонента в соответствии с его функциональным назначением, для этого необходимо изменить свойство Name. Ниже в табл. 2.2 приведены основные свойства компонента TextBox. Поскольку в программе необходимо ввести данные о коэффициентах квадратного уравнения, то нам потребуется еще два компонента TextBox. Добавьте недостающие компоненты TextBox самостоятельно, чтобы ваша форма выглядела так, как показано на рис. 2.8.

Рис. 2.7 ▼ Теперь форма содержит компонент TextBox

Рис. 2.8 ▼ Форма приложения содержит три компонента `TextBox`

Таблица 2.2 ▼ Основные свойства компонента `TextBox`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Text`	Текст, который содержится в поле для редактирования
`Font`	Шрифт, который используется для отображения текста
`ForeColor`	Цвет текста, находящегося в поле компонента
`BackColor`	Цвет фона поля компонента
`Location.X`	Расстояние от левой границы формы до левой границы компонента
`Location.Y`	Расстояние от верхней границы формы до верхней границы компонента
`Size.Width`	Ширина поля компонента
`Size.Height`	Высота поля компонента
`BorderStyle`	Вид рамки компонента. По умолчанию задается обычная (`Fixed3D`) рамка. Свойство также может принимать значения `FixedSingle` (тонкая рамка) и `None` (рамка отсутствует)
`TextAlign`	Способ выравнивания текста в поле компонента. Текст может быть прижат к левому краю (`Left`), правому краю (`Right`) или быть выровненным по центру (`Center`)

Значения некоторых свойств компонента можно изменять также с помощью мыши. Мышью можно задавать положение компонента на форме, а также его размер. Все вышесказанное также относится и к форме. Для того чтобы изменить положение компонента,выполните следующие действия:

1. Установите курсор мыши на его изображении.
2. Нажмите левую кнопку мыши и, не отпуская ее, переместите контур компонента в нужную точку формы.
3. Отпустите левую кнопку мыши.

Аналогично можно изменять размер компонента:

1. Щелкните по изображению компонента.
2. Установите курсор мыши на один из маркеров, размещенных по контуру компонента.
3. Нажмите левую кнопку мыши и, не отпуская ее, измените положение границы компонента.
4. Отпустите левую кнопку мыши.

В табл. 2.3 приведены значения свойств компонентов `TextBox1`, `TextBox2` и `TextBox3`, которые нам необходимо изменить. Значения остальных свойств мы менять не будем, и поэтому они в таблице не рассмотрены. Несколько поясню наличие пустой строки в свойстве `Text`. По умолчанию в этом поле было значение `TextBox1` (`TextBox2`, `TextBox3`), однако так как мы планируем вводить числовые значения, то текстовую информацию желательно удалить (или же, если сказать по-другому, записать в свойство `Text` «пустую» строку).

Таблица 2.3 ▾ Значения свойств компонентов `TextBox1`–`TextBox3`

Свойство	TextBox1	TextBox2	TextBox3
`Text`			
`Location.X`	40	40	40
`Location.Y`	48	76	104
`Size.Width`	48	48	48

Компонент `TextBox1` необходим нам для ввода коэффициента при второй степени, `TextBox2` – при первой, `TextBox3` – при нулевой. После того как значения для этих компонентов будут определены, ваша форма будет выглядеть так, как показано на рис. 2.9. Обратите внимание, что свойства `Font` всех трех компонентов соответствуют шрифту, который мы задавали для формы.

Кроме полей для ввода нам необходимо отображать поясняющий к ним текст, а также выводить результат. Для этих целей используют компонент `Label` – компонент для отображения текста. Этот компонент также находится на вкладке **Windows Forms** окна **Tool Palette**. Добавляется компонент на форму тем же способом, что и `TextBox`. Чтобы не заострять внимание на перенесении компонентов, скажу сразу, что все они перемещаются на форму одинаково. Замечу, что такая технология получила название «технологии визуального

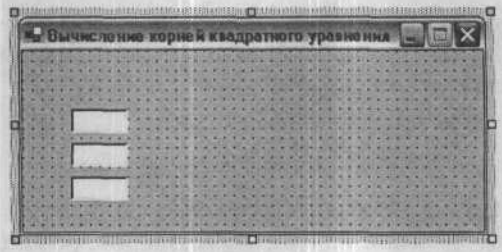

Рис. 2.9 ▼ Вид формы приложения после задания свойств компонентов TextBox1–TextBox3

проектирования» – программист создает и сразу наблюдает внешний вид программы на экране, при этом в окне редактора кода автоматически создается текст программы.

Для настройки компонента `Label` ознакомимся с его основными свойствами (табл. 2.4).

Таблица 2.4 ▼ Основные свойства компонента `Label`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Text`	Отображаемый в поле компонента текст
`Font`	Шрифт, который используется для отображения текста
`ForeColor`	Цвет текста, находящегося в поле компонента
`BackColor`	Цвет фона поля компонента
`Location.X`	Расстояние от левой границы формы до левой границы компонента
`Location.Y`	Расстояние от верхней границы формы до верхней границы компонента
`Size.Width`	Ширина поля компонента
`Size.Height`	Высота поля компонента
`BorderStyle`	Вид рамки компонента. По умолчанию задается обычная (`Fixed3D`) рамка. Свойство также может принимать значения `FixedSingle` (тонкая рамка) и `None` (рамка отсутствует)
`TextAlign`	Способ выравнивания текста в поле компонента. Текст может быть прижат к левому краю (`Left`), правому краю (`Right`) или быть выровненным по центру (`Center`)

После того как мы получили представление об этом компоненте, нам остается лишь определиться, какую информацию необходимо отображать. Нам понадобится отобразить фразу, предлагающую ввести коэффициенты квадратного

уравнения, подписи к полям для ввода коэффициентов и, собственно, результаты расчета, – итого пять текстовых строк. Соответственно, понадобится добавить на форму пять компонентов Label. Добавьте необходимые компоненты на форму и установите значения их свойств в соответствии с табл. 2.5.

Таблица 2.5 ▾ Свойства компонентов Label1–Label5

Свойство	Label1	Label2	Label3	Label4	Label5
Text	Введите значения коэффициентов квадратного уравнения	A:	B:	C:	
Location.X	24	24	24	24	104
Location.Y	16	55	83	111	48
Size.Width	328	16	16	16	120
Size.Height	24	16	16	16	80

После этого ваша форма будет выглядеть так, как показано на рис. 2.10, – уже нечто похожее на нужное нам окно.

Рис. 2.10 ▾ Вид формы после настройки компонентов Label1–Label4

Последнее, что необходимо сделать, – добавить на форму две кнопки, с помощью которых мы будем выполнять расчет и завершать программу. Для помещения кнопки на форму необходимо воспользоваться компонентом Button. Ниже в табл. 2.6 приведены основные свойства этого компонента, а в табл. 2.7 приведены значения свойств компонентов Button1 и Button2.

Таблица 2.6 ▾ Основные свойства компонента Button

Свойство	Комментарий
Name	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
Text	Отображаемый на кнопке текст

Таблица 2.6 ▼ Основные свойства компонента `Button` (окончание)

Свойство	Комментарий
`Font`	Шрифт, используемый для отображения текста на кнопке
`ForeColor`	Цвет текста, отображаемого на кнопке
`Location.X`	Расстояние от левой границы формы до левой границы компонента
`Location.Y`	Расстояние от верхней границы формы до верхней границы компонента
`Size.Width`	Ширина кнопки
`Size.Height`	Высота кнопки
`Enabled`	Признак доступности кнопки. Если значение свойства равно `True`, то кнопка доступна, если `False` – кнопка недоступна
`Image`	Свойство задает картинку на поверхности кнопки

Таблица 2.7 ▼ Свойства компонентов `Button1` и `Button2`

Свойство	Button1	Button2
`Text`	Вычислить	Завершение работы
`Location.X`	240	240
`Location.Y`	48	96
`Size.Width`	120	120
`Size.Height`	32	32

Завершив работу над формой, мы добились нужного внешнего вида окна и теперь можно приступать к написанию программы. Перед написанием программы остановимся на двух важных понятиях – событие и процедура обработки события.

Немного о том, как программируется поведение программы

На прошлом этапе мы настроили форму, определяющую внешний вид окна при выполнении программы. Если посмотреть на эту форму, то становится очевидным, что пользователь должен ввести в поля для редактирования исходные данные и нажать кнопку **Вычисление корней**. При нажатии на эту кнопку возникнет событие, которое отслеживается в Windows.

Событие (Event) – это то, что происходит во время выполнения программы. Каждое событие имеет имя. Например, щелчок по кнопке **Вычисление корней** вызывает событие `Click` (Нажатие). События не обязательно связаны со щелчками по кнопкам мыши – они также возникают при нажатии клавиш клавиатуры, перемещении мыши и т.д. Ниже в табл. 2.8 приведены основные события, которые происходят во время выполнения программы.

Таблица 2.8 ▾ Основные события, возникающие при работе программ

Событие	Момент возникновения события
`Click`	При щелчке кнопкой мыши
`DblClick`	При двойном щелчке кнопкой мыши
`MouseDown`	При нажатии кнопки мыши
`MouseUp`	При отпускании кнопки мыши
`MouseMove`	При перемещении указателя мыши
`KeyDown`	При нажатии клавиши. Сразу за данным событием следует событие `KeyPress`. При удерживании нажатой клавиши эта пара событий повторяется, пока не будет отжата клавиша. При отжатии клавиши формируется событие `KeyUp`
`KeyPress`	При нажатии клавиши сразу за событием `KeyDown`
`KeyUp`	При отпускании нажатой ранее клавиши
`Paint`	При появлении окна программы в начале работы, при появлении части окна, которая была закрыта другими окнами, при активизации окна программы из неактивного состояния
`Enter`	При получении элементом управления фокуса
`Leave`	При потере элементом управления фокуса
`Load`	В момент загрузки формы
`Closing`	При нажатии на системную кнопку **Закрыть** программы
`Closed`	Сразу после события `Closing`

При возникновении события программе необходимо знать, что нужно сделать, – соответственно реакцией на событие должно быть какое-то действие программы. Следует заметить, что число событий достаточно велико, поэтому нет необходимости описывать *все* события – описывать необходимо только те, на которые должна реагировать программа (в нашем случае программа должна среагировать только на два события `Click` – нажатие на кнопку **Вычислить** и **Завершение работы** соответственно). В Delphi такая реакция называется *процедурой обработки события* (Event handler). Таким образом, чтобы программа знала, какие действия ей выполнять при возникновении определенного события, программист должен написать такую процедуру. Ниже будет подробно рассмотрен перечень действий при разработке процедуры обработки события `Click` для кнопки **Вычислить**.

Чтобы приступить к созданию процедуры обработки события, необходимо:

1. Выбрать компонент, для которого мы будем писать процедуру (в нашем случае это `Button1`, то есть кнопка **Вычислить**).
2. В окне **Object Inspector** выбрать вкладку **Events**.

3. В левой колонке перечислены события, которые могут быть восприняты этим компонентом; как видите, их количество достаточно велико. Выберите нужное нам событие (`Click`), возникающее при нажатии кнопки.
4. Введите имя процедуры обработки события, например `Calculate`, и нажмите клавишу **Enter**.

В результате в программу будет вставлена процедура обработки события и станет активным окно редактора кода, в котором нам предстоит воспользоваться нашими знаниями языка Delphi (рис. 2.11).

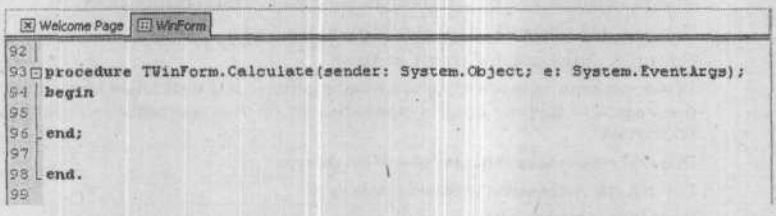

Рис. 2.11 ▼ Создание процедуры обработки события `Click`

Можно создать процедуру обработки и другим способом – достаточно выбрать нужное событие и выполнить двойной щелчок мышью вместо ввода имени процедуры. В этом случае имя процедуры будет сгенерировано автоматически – первая часть имени идентифицирует компонент, вторая – событие. Например, для нашего случая будет создана процедура с именем `TWinForm.Button1_Click`.

Еще один способ создания обработчика события – это двойной щелчок по самому компоненту. В этом случае среда сформирует обработчик события по умолчанию и привяжет его к нужному событию. Например, для кнопки событием по умолчанию является событие `Click`. Как и в предыдущем случае, имя обработчика события будет сформировано аналогичным образом.

У процедуры обработки имеется два параметра. Первый – `Sender` – идентифицирует объект, вызвавший событие. Второй – `EventArgs` – используется для передачи в процедуру обработки информации о событии. Какая информация может быть передана в процедуру? При нажатии клавиши, например, пересылается код нажатой клавиши. При обработке движения указателя мыши могут передаваться его координаты. Соответственно, набор передаваемых параметров зависит от типа компонента и типа события.

В листинге 2.1 приведен текст процедуры обработки события `Click` на кнопке **Вычислить** (`Button1`). Приведенный листинг показывает, как отображается

текст программы в редакторе кода, – ключевые слова выделяются полужирным шрифтом, комментарии – курсивом.

Листинг 2.1 ▾ Процедура обработки события `Click`

```
procedure TWinForm1.Button1_Click(sender:System.Object;
                                  e: System.EventArgs);
var
  A,B,C: single;
  D: real;
  X1,X2: real;
begin
  // Ввод исходных данных.
  A := System.Convert.ToSingle(TextBox1.Text);
  B := System.Convert.ToSingle(TextBox2.Text);
  C := System.Convert.ToSingle(TextBox3.Text);
  // Вычисление дискриминанта.
  D := B*B-4*A*C;
  // Вычисление корней уравнения.
  if D < 0 then
    // Если дискриминант отрицателен, то выводим сообщение
    // о том, что уравнение не имеет корней.
    begin
      Label5.Text := 'Уравнение не имеет действительных корней';
    end
    // Если дискриминант больше или равен 0, то
    // вычисляем корни уравнения.
    else
      begin
        X1 := (-B-sqrt(D))/(2*A);
        X2 := (-B+sqrt(D))/(2*A);
        // Выводим рассчитанные значения корней.
       Label5.Text := 'X1 = '+X1.ToString+', X2 = '+X2.ToString;
      end;
end;
```

Теперь разберем приведенный в листинге исходный код.

Процедура обработки `TWinForm.Button1_Click` выполняет вычисление корней квадратного уравнения и выводит результат в метку `Label5`. Исходные данные о коэффициентах при неизвестном вводятся из полей редактирования `TextBox1`, `TextBox2` и `TextBox3`. После ввода данных пользователем значение свойств `Text` этих компонентов меняется на те, что ввел пользователь. Для преобразования введенных пользователем строк в числа используется функция `ToSingle`, которая входит в пространство имен

System.Convert (о пространстве имен мы подробно поговорим несколько позднее). Результатом этой функции является число, полученное из введенной пользователем строки. Поэтому если пользователь введет какой-либо текст, который невозможно перевести в число, то будет выдано сообщение об ошибке. Таким образом, функция ToSingle сработает только в том случае, если передаваемое число записано в верном формате, то есть предполагается ввод целого числа либо дробного, с использованием запятой в качестве разделителя.

Немного о пространстве имен. *Пространство имен* – это модуль, предоставляющий использующей его программе свои объекты (функции, константы и т.п.). Во время написания программы Delphi предоставляет нам свои модули различного назначения. Например, при создании формы мы использовали компоненты типа TextBox, Label и Button. Все эти компоненты содержатся в модуле (пространстве имен) System.Windows.Forms. В противном случае нам пришлось бы самим писать исходный код для подобных объектов. Так что наша задача существенно упрощается, так как программисты Borland предоставили нам возможность использовать все то, что было написано до нас (в самом деле, совершенно незачем заниматься изобретением велосипеда). Таким образом, любой объект относится к тому или иному пространству имен.

Пространства имен, которые использует программа, перечислены в секции Uses (см. окно редактора кода). Пространство имен System доступно по умолчанию и поэтому в этой секции указывать его нет необходимости.

Вычисленные значения корней выводятся в поле Label5 путем присваивания значения свойству Text. Для обратного преобразования числа в строку используется метод ToString. Его параметром является символьная константа, характеризующая формат отображения числа. Функция ToString может использоваться и без параметра.

Ниже в табл. 2.9 приведены основные форматы отображения чисел.

Таблица 2.9 ▼ Основные форматы отображения чисел

Константа	Формат отображения	Пример отображения
c, C	Денежный формат – Currency. Используется для отображения денежных величин. При использовании этого формата отображения необходимо учитывать, что обозначение денежной единицы, разделитель групп разрядов и способ отображения отрицательных чисел определяются настройками операционной системы	1200,54 р.
f, F	Число с фиксированной точкой. Служит для представления дробных чисел. Количество цифр дробной части, а также символ-разделитель целой и дробной частей определяются настройками операционной системы	1200,54

Таблица 2.9 ▾ Основные форматы отображения чисел (окончание)

Константа	Формат отображения	Пример отображения
e, E	Научный формат – Exponential. Применяется для представления очень маленьких или очень больших чисел	1,20054+E03
n, N	Числовой формат – Number. Применяется для представления дробных чисел. При оставлении нужного количества дробных цифр производится округление. Параметры отображения также зависят от настроек операционной системы	1 200,54
g, G	Универсальный (основной) формат – General. Практически то же, что и Number, однако разряды в данном формате не разделяются пробелами	1200,542
r, R	Формат, не производящий округления – RoundTrip. В отличие от формата Number не производит округления	1200,5422

Как видно из табл. 2.9, отображение чисел зависит от настроек операционной системы. Поэтому при создании своих программ следует воспользоваться специальными объектами, устанавливающими правила отображения чисел на момент выполнения вашей программы. Подобным способом вы предостережете себя от зависимости настроек операционной системы, на которой будет выполняться ваша программа, – числа будут отображаться всегда одинаково в соответствии с вашими настройками. Поэтому советую запустить справочную систему Borland Delphi 2005 и воспользоваться сведениями об объектах, устанавливающих правила отображения чисел:

- для формата Currency – CurrencyNegativePattern, CurrencyPositivePattern, CurrencySymbol, CurrencyGroupSize, CurrencyGruopSeparator, CurrencyDecimalDigits, CurrencyDecimalSeparator;
- для формата Number – NumberNegativePattern, NumberGroupSize, NumberGroupSeparator, NumberDecimalDigits, NumberDecimal Separator.

Работаем в редакторе кода

Во время набора текста программы редактор автоматически выделяет элементы программы: полужирным – ключевые слова языка программирования (procedure, begin, end и т.п.); курсивом – комментарии; цветом – строковые константы. Все это делает исходный код более выразительным, а также облегчает восприятие структуры программы.

В процессе разработки программы часто возникает необходимость переключения между окном редактора кода и окном формы. Выбрать нужное окно можно нажатием на соответствующую вкладку **Code** или **Design** (рис. 2.12) либо нажатием клавиши **F12**.

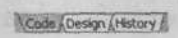

Рис. 2.12 ▾ Переключение между окнами редактора кода и дизайнера формы

Используем систему подсказок

Во время набора кода редактор постоянно отслеживает то, что вы набираете. При этом он старается всячески облегчить ваш труд, помогая ускорить этот процесс. Например, если вы набираете имя какого-либо объекта и затем ставите точку, то редактор кода автоматически выведет вам список свойств и методов объекта (рис. 2.13). Вам же останется лишь выбрать из списка необходимый элемент и нажать клавишу **Enter**. Важно также заметить, что если при тех же действиях такой список не появляется, – значит, в программе обнаружена ошибка (например, вы набрали неправильное имя компонента или служебное слово языка Delphi).

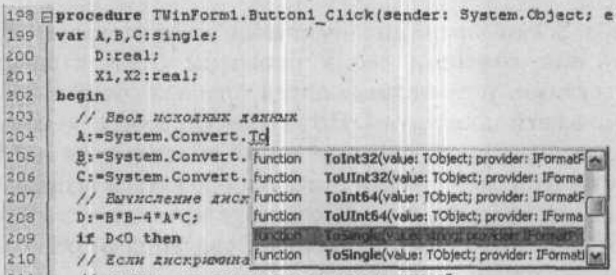

Рис. 2.13 ▼ Всплывающие подсказки во время набора кода

Кроме того, если после того как вы ввели точку и появился список с подсказкой начнете вводить имя метода или свойства, то среда автоматически будет фильтровать доступные варианты. Также есть способ вызвать это окно подсказки в любое время. Для этого нажмите комбинацию клавиш **Ctrl+Пробел**.

При использовании функций (встроенных либо написанных программистом) редактор кода также выдает подсказку – список параметров для данной функции.

Применяем шаблоны кода

В процессе набора текста иногда бывает необходимо использовать шаблоны кода. Что это такое? *Шаблон кода* – это инструкции языка, отражающие структуру будущей программы (более подробную информацию об инструкциях языка Delphi вы найдете в главе 3). Например, шаблон кода для инструкции обработки возможных ошибок выглядит следующим образом:

```
try
except
end;
```

Смысл использования шаблонов состоит в ускоренном наборе текста программ с помощью заготовок для сложных операторов.

В редакторе кода предусмотрено большое количество шаблонов, в том числе и несколько типов для некоторых инструкций (например, для инструкций `if` или `while`).

Для того чтобы воспользоваться шаблоном кода, следует нажать комбинацию клавиш **Ctrl+J** и из появившегося списка выбрать необходимый шаблон (рис. 2.14). Программист может создать собственный шаблон, но эту функцию пока рассматривать не будем.

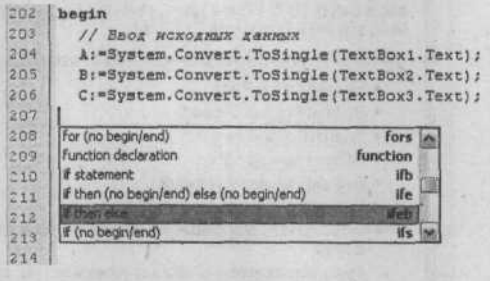

Рис. 2.14 ▾ Использование шаблонов исходного кода

Получаем оперативную справочную информацию

В процессе набора текста программы нам могут помочь не только шаблоны кода и всплывающие подсказки. В самом худшем варианте, когда мы вообще не знаем, как решить ту или иную проблему, стоит воспользоваться справочной системой. Для получения справки достаточно, находясь в редакторе кода, нажать клавишу **F1**. Информация, которая вам будет показана, зависит от того, где располагался текстовый курсор на момент нажатия клавиши **F1**. Соответственно, будет выведена информация о том объекте, на котором располагался текстовый курсор. Справочную систему можно вызвать и другим способом – выбрать пункт **Borland Help** в меню **Help**. В этом случае будет показано стандартное окно справочной системы (рис. 2.15). В этом окне можно воспользоваться самораскрывающимся списком **Look for** на вкладке **Index**. В этом списке можно ввести ключевое слово, определяющее, что вы хотите найти. В качестве ключевого слова можно ввести, например, несколько первых букв имени функции, свойства, метода и т.п.

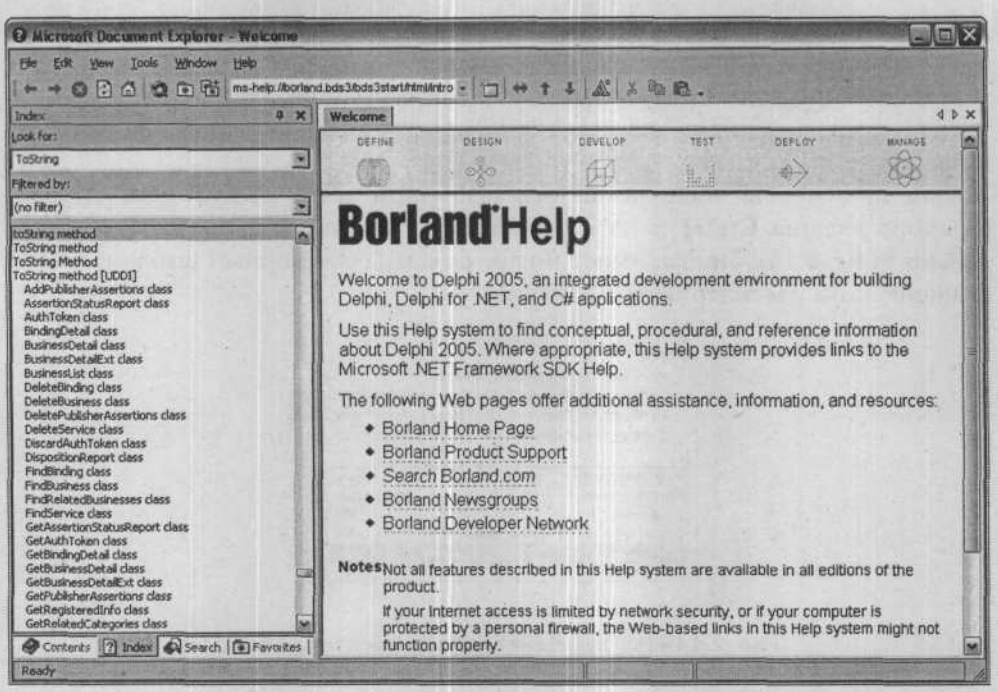

Рис. 2.15 ▼ Окно справочной системы Borland Delphi 2005

Изучаем структуру проекта для .NET

Вернемся снова к нашему проекту – уже практически все готово, и мы находимся в двух шагах от завершения нашей программы. Мы создали форму, а также составили исходный код, описывающий, как будет работать программа. Что дальше? А дальше требуется *сохранить* на диске все наши наработки и *откомпилировать* программу, чтобы получить заветный исполняемый exe-файл. Но, прежде чем мы это сделаем, нам необходимо разобраться, а что же мы на данном этапе создали и как хранится информация о нашей форме, коде и т.п.

Как мы уже знаем, вся информация, необходимая для получения конечного exe-файла, в Delphi называется проектом. Реально проект в простейшем случае представляет собой совокупность следующих файлов:

- файл описания проекта (bdsproj-файл);
- главный модуль (dpr-файл);

- модуль формы (pas-файл);
- файл ресурсов (res-файл);
- файл конфигурации (cfg-файл);
- файл ресурсов формы (resx-файл).

Файл описания проекта

Файл описания проекта (bdsproj-файл) представляет собой файл специального формата, в котором записана общая информация о проекте: его состав, название, настройки компилятора и т.д. Этот файл формируется автоматически и вносить в него изменения следует только в крайних случаях. Нам же, как начинающим программистам, это вряд ли понадобится, поэтому мы для себя отметим его существование, но редактировать его не будем.

Главный модуль

Главный модуль (dpr-файл) содержит инструкции, обеспечивающие запуск нашей программы, а также команды для компилятора и необходимую информацию для формирования так называемой *сборки*. Сборкой в .NET называются модули, в том числе и откомпилированная программа. Главный модуль также автоматически формируется средой Delphi. В качестве примера в листинге 2.2 приведен главный модуль нашей программы. Для того чтобы открыть этот файл в редакторе кода, в главном меню выберите **View ➤ Units** и в появившемся диалоге дважды щелкните по файлу **Project1**. Есть еще один способ открыть файл проекта. В окне **Project Manager** щелкните правой кнопкой мыши по имени проекта (напомню, что в нашем случае это **Project1**) и в появившемся меню выберите пункт **View Source**.

Листинг 2.2 ▾ Главный модуль программы

```
program Project1;
{%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
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.XML.dll'}
{$R 'WinForm1.TWinForm1.resources' 'WinForm1.resx'}
uses
```

```
  System.Reflection,
  System.Runtime.CompilerServices,
  System.Windows.Forms,
  WinForm1 in 'WinForm1.pas' {WinForm1.TWinForm1:
                              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('')]
{$ENDREGION}
[STAThread]
begin
  Application.Run(TWinForm1.Create);
end.
```

В заголовке модуля указано имя программы, далее следуют команды, которые содержат информацию о том, что именно (какие модули, библиотеки и т.п.) понадобится компилятору Delphi для выполнения сборки нашего проекта. Как видите, для сборки нам необходимы динамические библиотеки (dll-файлы), которые входят в состав Microsoft .NET Framework (помните, мы устанавливали этот компонент в самом начале?). Теперь с уверенностью можно сказать, для чего нам был нужен компонент Microsoft .NET framework, – он содержит все необходимые модули, в которых описаны все используемые нами функции, объекты, компоненты и т.п. Посмотрите внимательно на листинг – вы увидите, что все то, что мы видели в окне редактора кода в секции `uses`, используется для составления перечня необходимых для выполнения сборки компонентов. Теперь перейдем к подробному рассмотрению модуля формы.

Модуль формы

Модуль формы содержит информацию о ее настройках, компонентах, которые на ней присутствуют, а также процедуры обработки событий для этих компонентов. Большая часть файла также формируется с помощью среды Delphi

автоматически. В листинге 2.3 приведен текст модуля формы нашей программы. Теперь осталось разобраться, какая часть текста формируется автоматически, а какая создается непосредственно программистом.

Листинг 2.3 ▾ Модуль формы программы

```
unit WinForm1;

interface

uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data;

type
  TWinForm1 = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    Components: System.ComponentModel.Container;
    TextBox1: System.Windows.Forms.TextBox;
    TextBox2: System.Windows.Forms.TextBox;
    TextBox3: System.Windows.Forms.TextBox;
    Label1: System.Windows.Forms.Label;
    Label2: System.Windows.Forms.Label;
    Label3: System.Windows.Forms.Label;
    Label4: System.Windows.Forms.Label;
    Label5: System.Windows.Forms.Label;
    Button1: System.Windows.Forms.Button;
    Button2: System.Windows.Forms.Button;
    procedure InitializeComponent;
    procedure Button2_Click(sender: System.Object;
                            e: System.EventArgs);
    procedure Button1_Click(sender: System.Object;
                            e: System.EventArgs);
  {$ENDREGION}
  strict protected
    procedure Dispose(Disposing: Boolean); override;
  private
    { Private Declarations }
  public
    constructor Create;
  end;
  [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm1))]

implementation

{$AUTOBOX ON}
{$REGION 'Windows Form Designer generated code'}
```

```
procedure TWinForm1.InitializeComponent;
begin
  Self.TextBox1 := System.Windows.Forms.TextBox.Create;
  Self.TextBox2 := System.Windows.Forms.TextBox.Create;
  Self.TextBox3 := System.Windows.Forms.TextBox.Create;
  Self.Label1 := System.Windows.Forms.Label.Create;
  Self.Label2 := System.Windows.Forms.Label.Create;
  Self.Label3 := System.Windows.Forms.Label.Create;
  Self.Label4 := System.Windows.Forms.Label.Create;
  Self.Label5 := System.Windows.Forms.Label.Create;
  Self.Button1 := System.Windows.Forms.Button.Create;
  Self.Button2 := System.Windows.Forms.Button.Create;
  Self.SuspendLayout;
  //
  // TextBox1
  //
  Self.TextBox1.Location := System.Drawing.Point.Create(40, 48);
  Self.TextBox1.Name := 'TextBox1';
  Self.TextBox1.Size := System.Drawing.Size.Create(48, 20);
  Self.TextBox1.TabIndex := 0;
  Self.TextBox1.Text := '';
  //
  // TextBox2
  //
  Self.TextBox2.Location := System.Drawing.Point.Create(40, 76);
  Self.TextBox2.Name := 'TextBox2';
  Self.TextBox2.Size := System.Drawing.Size.Create(48, 20);
  Self.TextBox2.TabIndex := 1;
  Self.TextBox2.Text := '';
  //
  // TextBox3
  //
  Self.TextBox3.Location := System.Drawing.Point.Create(40, 104);
  Self.TextBox3.Name := 'TextBox3';
  Self.TextBox3.Size := System.Drawing.Size.Create(48, 20);
  Self.TextBox3.TabIndex := 2;
  Self.TextBox3.Text := '';
  //
  // Label1
  //
  Self.Label1.Location := System.Drawing.Point.Create(24, 16);
  Self.Label1.Name := 'Label1';
  Self.Label1.Size := System.Drawing.Size.Create(328, 24);
```

```
Self.Label1.TabIndex := 3;
Self.Label1.Text :=
        'Введите значения коэффициентов квадратного уравнения:';
//
// Label2
//
Self.Label2.Location := System.Drawing.Point.Create(24, 55);
Self.Label2.Name := 'Label2';
Self.Label2.Size := System.Drawing.Size.Create(16, 16);
Self.Label2.TabIndex := 4;
Self.Label2.Text := 'A:';
//
// Label3
//
Self.Label3.Location := System.Drawing.Point.Create(24, 83);
Self.Label3.Name := 'Label3';
Self.Label3.Size := System.Drawing.Size.Create(16, 16);
Self.Label3.TabIndex := 5;
Self.Label3.Text := 'B:';
//
// Label4
//
Self.Label4.Location := System.Drawing.Point.Create(24, 111);
Self.Label4.Name := 'Label4';
Self.Label4.Size := System.Drawing.Size.Create(16, 16);
Self.Label4.TabIndex := 6;
Self.Label4.Text := 'C:';
//
// Label5
//
Self.Label5.BorderStyle :=
                    System.Windows.Forms.BorderStyle.Fixed3D;
Self.Label5.Location := System.Drawing.Point.Create(104, 48);
Self.Label5.Name := 'Label5';
Self.Label5.Size := System.Drawing.Size.Create(120, 80);
Self.Label5.TabIndex := 7;
//
// Button1
//
Self.Button1.Location := System.Drawing.Point.Create(240, 48);
Self.Button1.Name := 'Button1';
Self.Button1.Size := System.Drawing.Size.Create(120, 32);
Self.Button1.TabIndex := 8;
```

```
  Self.Button1.Text := 'вычислить';
  Include(Self.Button1.Click, Self.Button1_Click);
  //
  // Button2
  //
  Self.Button2.Location := System.Drawing.Point.Create(240, 96);
  Self.Button2.Name := 'Button2';
  Self.Button2.Size := System.Drawing.Size.Create(120, 32);
  Self.Button2.TabIndex := 9;
  Self.Button2.Text := 'завершение работы';
  Include(Self.Button2.Click, Self.Button2_Click);
  //
  // TWinForm1
  //
  Self.AutoScaleBaseSize := System.Drawing.Size.Create(5, 13);
  Self.ClientSize := System.Drawing.Size.Create(384, 150);
  Self.Controls.Add(Self.Button2);
  Self.Controls.Add(Self.Button1);
  Self.Controls.Add(Self.Label5);
  Self.Controls.Add(Self.Label4);
  Self.Controls.Add(Self.Label3);
  Self.Controls.Add(Self.Label2);
  Self.Controls.Add(Self.Label1);
  Self.Controls.Add(Self.TextBox3);
  Self.Controls.Add(Self.TextBox2);
  Self.Controls.Add(Self.TextBox1);
  Self.FormBorderStyle :=
                  System.Windows.Forms.FormBorderStyle.FixedSingle;
  Self.MaximizeBox := False;
  Self.Name := 'TWinForm1';
  Self.StartPosition :=
                System.Windows.Forms.FormStartPosition.CenterScreen;
  Self.Text := 'Вычисление корней квадратного уравнения';
  Self.ResumeLayout(False);
end;
{$ENDREGION}

procedure TWinForm1.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
```

```
  inherited Dispose(Disposing);
end;
constructor TWinForm1.Create;
begin
  inherited Create;
  InitializeComponent;
end;
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var A,B,C:single;
    D:real;
    X1,X2:real;
begin
  // Ввод исходных данных.
  A := System.Convert.ToSingle(TextBox1.Text);
  B := System.Convert.ToSingle(TextBox2.Text);
  C := System.Convert.ToSingle(TextBox3.Text);
  // Вычисление дискриминанта.
  D := B*B-4*A*C;
  if D<0 then
  // Если дискриминант отрицателен, то выводим сообщение
  // о том, что уравнение не имеет корней.
    begin
    Label5.Text := 'Уравнение не имеет действительных корней';
    end
  // Если дискриминант больше или равен 0, то
  // вычисляем корни уравнения.
  else
    begin
    X1 := (-B-sqrt(D))/(2*A);
    X2 := (-B+sqrt(D))/(2*A);
    // Выводим рассчитанные значения корней.
    Label5.Text := 'X1 = '+X1.ToString+#13+'X2 = '+X2.ToString;
    end;
end;
procedure TWinForm1.Button2_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
  // Завершаем работу программы.
  Close;
end;
end.
```

Модуль формы можно условно разделить на две части – раздел интерфейса и раздел реализации.

Секция интерфейса (начинается со служебного слова `interface`) содержит автоматически сформированное Delphi-объявление типа формы. Она также содержит описание компонентов и процедур (но не сами процедуры!) обработки событий. Как формируется эта секция? Во время создания нашей формы мы переносили на нее некоторые компоненты. При этом «перетаскивание» компонента на форме приводит к тому, что среда Delphi сама описывает компонент, который вы перетащили на форму, и устанавливает те начальные свойства, которые вы прописывали в окне **Object Inspector** для данного компонента. Когда программирование было не настолько сильно развито, то программисту приходилось писать все эти строки вручную, что, естественно, сказывалось на времени разработки программ. Поэтому вы, конечно, можете вносить изменения, например, в свойства компонентов прямо в окне редактора кода, но делать это гораздо проще и удобнее через окно **Object Inspecor**.

Секция реализации начинается со служебного слова `implementation`. Она содержит объявление локальных процедур и функций, в том числе и процедур обработки событий. Как видите, программисту отводится в основном роль разработки внешнего вида программы, а также написание процедур обработки событий. Всю остальную рутинную работу выполняет среда программирования Delphi. Согласитесь, что такой подход удобен.

В принципе, существуют еще две секции – секция инициализации и финализации модуля. В нашем примере они отсутствуют. *Секция инициализации* (начинается со служебного слова `initialization` и следует за секцией реализации) выполняется при загрузке программы на выполнение, а *секция финализации* (начинается со служебного слова `finalization` и следует за секцией реализации), наоборот, – при завершении программы.

Сохраняем свои наработки

Теперь мы выяснили, что такое проект и какие файлы входят в его состав. Попробуем сохранить проект на диске нашего компьютера. Для этого в меню **File** выберите команду **Save All** (Сохранить весь проект). Перед вами появится диалоговое окно (рис. 2.16), в котором будет предложено сохранить модуль формы. Вам следует ввести имя модуля формы и нажать кнопку **Save** (Сохранить).

Модуль формы будет сохранен, и на экране появится следующее окно **Save Project** (Сохранение проекта), в котором надо будет ввести имя проекта

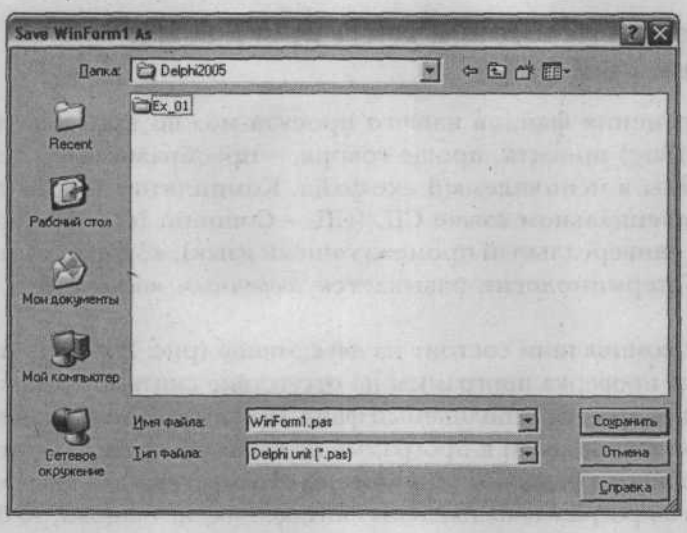

Рис. 2.16 ▾ Сохранение модуля формы

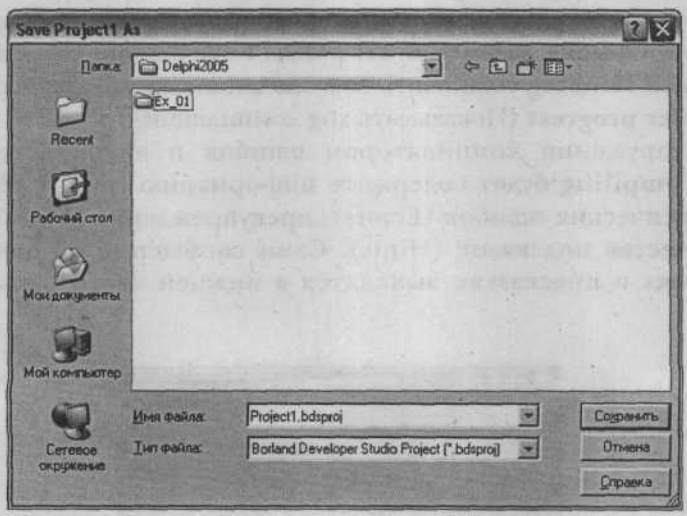

Рис. 2.17 ▾ Сохранение файла проекта

(рис. 2.17). Здесь следует обратить внимание на то, что имя генерируемого Delphi exe-файла совпадает с именем проекта.

Преобразуем исходный текст программы в исполняемый файл

После сохранения файлов нашего проекта можно приступать к *компиляции* (Compiling) проекта, проще говоря, – преобразованию исходного кода программы в исполняемый exe-файл. Компилятор Delphi создает программу на специальном языке CIL (CIL – Common Intermediate Language, дословно – универсальный промежуточный язык), которая согласно принятой в .NET терминологии называется *сборочным модулем*, или *сборкой* (Assembly).

Процесс компиляции состоит из двух этапов (рис. 2.19). На первом этапе выполняется проверка программы на отсутствие синтаксических ошибок, на втором – генерируется исполняемый файл. При этом второй этап выполняется только в том случае, если в программе не были обнаружены синтаксические ошибки. Процесс и результат компиляции отражается в диалоговом окне **Compiling**. Если в программе не найдены синтаксические ошибки, то будет выдано сообщение **Done: Compiled**. В противном случае будет выдано сообщение **Done: There are errors**.

Если окно компиляции не отображается, то необходимо выбрать в меню **Tools** команду **Options**, затем выбрать раздел **Environment Options** и в группе **Compiling and Running** установить во включенное состояние переключатель **Show compiler progress** (Показывать ход компиляции).

При обнаружении компилятором ошибок и неточностей диалоговое окно **Compiling** будет содержать информацию (рис. 2.18) о количестве синтаксических ошибок (Errors), предупреждений (Warnings), а также о количестве подсказок (Hints). Сами сообщения об ошибках, предупреждениях и подсказках выводятся в нижней части окна редактора кода.

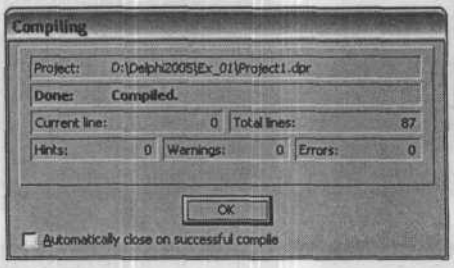

Рис. 2.18 ▼ Окно с результатами компиляции

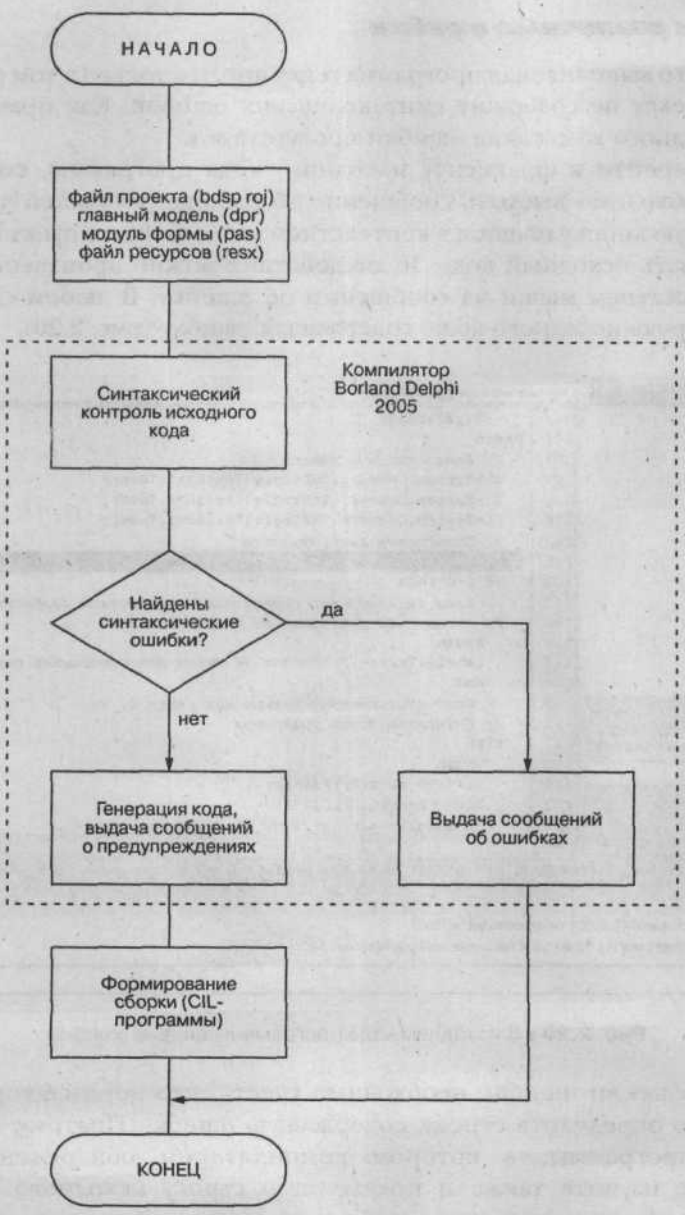

Рис. 2.19 ▾ Схема работы компилятора проекта

Устраняем различные ошибки

Напомню, что выполняемая программа генерируется только в том случае, если исходный текст не содержит синтаксических ошибок. Как правило, после набора исходного кода такие ошибки присутствуют.

Чтобы перейти к фрагменту исходного кода программы, содержащему ошибку, необходимо выбрать сообщение об ошибке (в нижней части окна), нажать правую кнопку мыши и в контекстном меню выбрать пункт **Edit Source** (Редактировать исходный код). Те же действия можно произвести простым двойным нажатием мыши на сообщении об ошибке. В любом случае будет выделена строка исходного кода, содержащая ошибку (рис. 2.20).

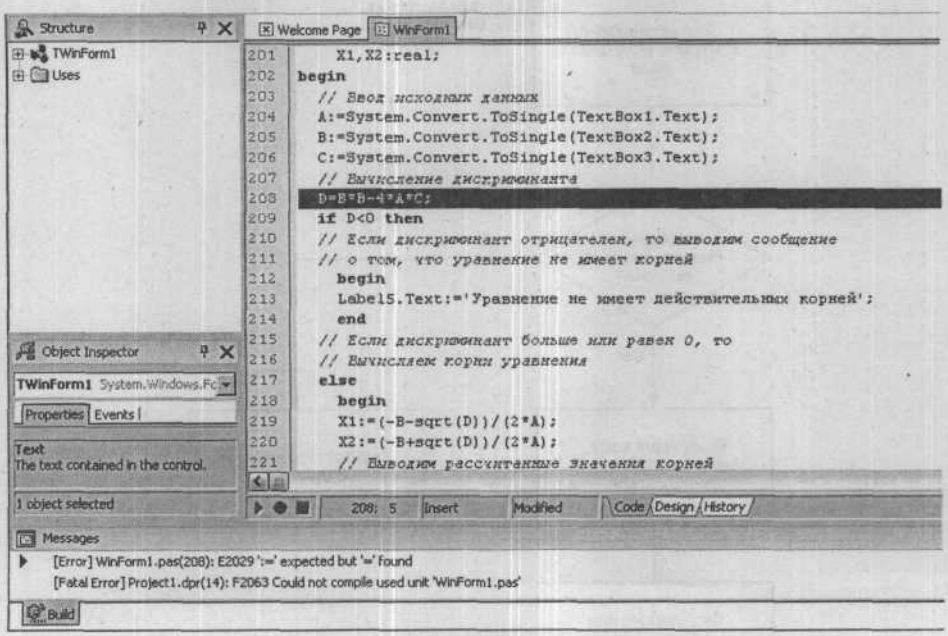

Рис. 2.20 ▼ В исходном коде программы найдена ошибка

При устранении ошибок необходимо учесть, что компилятор не всегда может точно определить строку, содержащую ошибку. Поэтому, анализируя фрагмент программы, в котором компилятором обнаружена ошибка, внимательно изучите также и предыдущую строку исходного кода. Если компилятор обнаружил много ошибок, то исправляйте сначала наиболее очевидные из них. Довольно часто случается так, что одна ошибка может

повлечь за собой другие, и при исправлении одной ошибки при следующей компиляции некоторые другие ошибки могут исчезнуть.

Если же ошибок нет, то среда Delphi создает exe-файл с именем проекта и помещает его в тот же каталог, что и каталог проекта.

На что следует обращать внимание – предупреждения и подсказки

При обнаружении неточностей, которые не являются ошибками, компилятор выводит предупреждения (Warnings) и подсказки (Hints). Ситуацией, при которой компилятор выводит предупреждение, может быть, например, объявление переменной, которая потом не будет использована в программе. Ошибки здесь никакой нет, но вам будет выдано сообщение о том, что переменная реально в программе не используется. Ситуации, при которых выводятся предупреждения, могут быть разными, поэтому делать изменения в исходном коде или нет, зависит от каждого конкретного случая. Однако обращать внимание на такие вещи все же стоит.

Запускаем полученную программу

Пробный запуск программы можно выполнить как из среды разработки, так и из Windows. В первом случае следует в меню **Run** выбрать пункт **Run**. При этом если вы сделаете какие-либо изменения в исходном коде программы, то она откомпилируется заново.

Во втором случае следует с помощью программы Проводник перейти к папке проекта и выполнить сформированный там в процессе компиляции exe-файл.

Подводные камни – ошибки, связанные с недоработкой программы

Теперь попробуйте в появившемся окне ввести необходимые данные и получить результат (рис. 2.21). Как видите, программа вычисляет корни уравнения.

Но действительно ли в нашей программе теперь нет ошибок? Попробуйте в полях для ввода указать ошибочные данные, например какую-нибудь строку или дробное число, в котором в качестве разделителя используется точка (по умолчанию дробные числа записываются через запятую)[1]. После этого

[1] Это зависит от настроек операционной системы. Если ваша программа Windows настроена на русский язык, то разделителем по умолчанию действительно будет запятая (если вы не поменяли эту настройку с помощью региональных установок). Для английского языка разделителем является точка. – *Прим. науч. ред.*

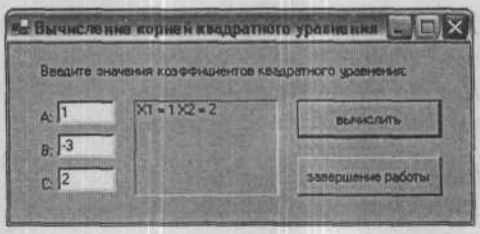

Рис. 2.21 ▼ Результат работы программы

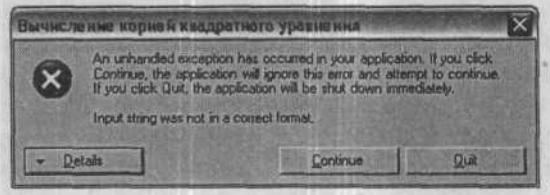

Рис. 2.22 ▼ Неправильные исходные данные привели к возникновению исключения (запуск из Windows)

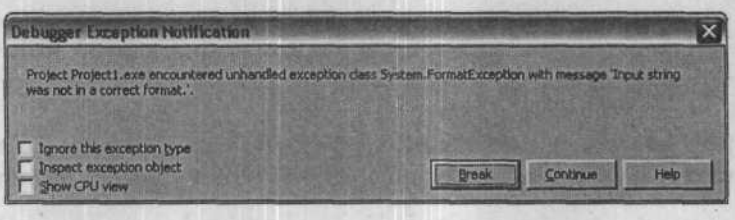

Рис. 2.23 ▼ Неправильные исходные данные привели к возникновению исключения (запуск из среды Borland Delphi 2005)

нажмите кнопку **Вычисление корней**, и перед вами появятся сообщения, показанные на рис 2.22. Если запуск был произведен из среды разработки Borland Delphi 2005, то сообщение будет несколько отличаться (рис. 2.23).

Наши действия привели к возникновению *исключения* (Exception), другими словами – произошла ошибка во время выполнения программы (чаще употребляют термин «ошибка времени выполнения программы»). Почему это произошло? Внимательно присмотритесь к сообщению на экране.

В нижней строке, которая указывает причину возникновения исключения, стоит надпись «Input string was not in correct format», что означает, что входные данные (в нашем случае это строка из поля ввода) представлены в неправильном формате. И действительно – из букв невозможно получить

цифры – функция преобразования строки в число ToSingle работает только тогда, когда в качестве ее параметра указывается изображение числа, разделенное запятой. Мы же ввели ошибочные данные, поэтому дальнейшая работа программы невозможна.

Необходимо отметить, что по умолчанию разделителем для дробных чисел является символ запятой. Однако если нам больше нравится точка или любой другой символ, то мы можем изменить настройки операционной системы (на время выполнения нашей программы). Символ-разделитель целой и дробной частей числа содержит объект CurrentCulture:

```
CurrentInfo.CurrentCulture.NumberFormat.NumberDecimalSeparator
```

Для завершения работы программы, в которой произошло исключение, необходимо закрыть окно сообщения об ошибке и в меню **Run** выбрать команду **Program Reset**.

Доработка программы на предмет возможных ошибок

Чтобы избежать появления сообщений об ошибках, которые никак не красят созданные нами программы, можно воспользоваться возможностями языка Delphi по обработке исключений. Другими словами, если мы можем предугадать ошибочные действия пользователя, то можем указать нашей программе, как вести себя в подобных ситуациях.

Исключения в Delphi обрабатываются следующей инструкцией:

```
try
  // Инструкции программы, которые могут вызвать исключение.
except
  on Исключение1 do
    begin
      // Инструкции обработки исключения.
    end;
...
  on ИсключениеN do
    begin
      // Инструкции обработки исключения.
    end;
end;
```

где:

try – ключевое слово, которое обозначает, что далее следуют инструкции, при выполнении которых может возникнуть исключение, обработку которых берет на себя программа;

except – ключевое слово, которое обозначает начало секции обработки исключений. В этой секции перечисляются исключения, обработку которых берет на себя программа.

Примеры наиболее часто возникающих исключений, а также вызывающих их причин приведены в табл. 2.10.

Таблица 2.10 ▼ Типовые исключения

Тип исключения	Причина возникновения
`FormatException`	Ошибка преобразования. Возникает при выполнении преобразования, если преобразуемая величина не может быть приведена к требуемому виду. Наиболее часты случаи при переводе чисел в строки и обратно
`OverflowException`	Ошибка переполнения. Возникает в результате выполнения операции при выходе за границы допустимого диапазона значения. Также возникает в случае операции деления, если делитель равен нулю
`FilenotFoundException`	Ошибка открытия файла (файл не найден). Возникает при выполнении файловых операций в том случае, если не удается найти необходимый файл на носителе

В разработанной нами программе возможно возникновение исключения `FormatException` (вспомните окно с сообщением об ошибке – рис. 2.23). Измененный текст программы, который учитывает возникновение подобной ошибки, приведен в листинге 2.4.

Листинг 2.4 ▼ Пример обработки исключения

```
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e:System.EventArgs);
var A,B,C:single;
    D:real;
    X1,X2:real;
begin
  // Ввод исходных данных.
  try
  A:=System.Convert.ToSingle(TextBox1.Text);
  B:=System.Convert.ToSingle(TextBox2.Text);
  C:=System.Convert.ToSingle(TextBox3.Text);
  except
  on e:FormatException do
    begin
    if (TextBox1.Text='')or(TextBox2.Text='')or(TextBox3.Text='')
then
    MessageBox.Show('Необходимо ввести все коэффициенты.',
    'Ошибка при вводе исходных данных!',MessageBoxButtons.OK,
    MessageBoxIcon.Information)
```

```
      else
      MessageBox.Show('При задании коэффициентов используйте
числа.',
      'Ошибка при вводе исходных данных!',MessageBoxButtons.OK,
      MessageBoxIcon.Information);
      exit;
      end;
    end;
    // Вычисление дискриминанта.
    D:=B*B-4*A*C;
    if D<0 then
    // Если дискриминант отрицателен, то выводим сообщение
    // о том, что уравнение не имеет корней.
      begin
      Label5.Text:='Уравнение не имеет действительных корней';
      end
    // Если дискриминант больше или равен 0, то
    // вычисляем корни уравнения.
    else
      begin
      X1:=(-B-sqrt(D))/(2*A);
      X2:=(-B+sqrt(D))/(2*A);
      // Выводим рассчитанные значения корней.
      Label5.Text:='X1 = '+X1.ToString+' X2 = '+X2.ToString;
      end;
  end;
```

Теперь при неправильном вводе данных пользователю будет выводиться диалоговое окно с сообщением об ошибке. Окно с сообщением выводится с помощью функции `MessageBox.Show`. В общем случае инструкция, обеспечивающая вывод подобного окна, выглядит следующим образом:

```
MessageBox.Show(<Сообщение>,<Заголовок>,<Кнопки>,<Иконка>);
```

Параметр <Кнопки> задает перечень отображаемых в выводимом окне кнопок. Перечень кнопок, которые возможно отобразить в данном окне, приведен в табл. 2.11.

Таблица 2.11 ▾ Константы, определяющие кнопки в окне сообщения

Константа, определяющая тип кнопки	Выводимые в окне сообщения кнопки
`MessageBoxButtons.Ok`	**OK**
`MessageBoxButtons.OkCancel`	**OK, Отмена**

Таблица 2.11 ▼ Константы, определяющие кнопки в окне сообщения (окончание)

Константа, определяющая тип кнопки	Выводимые в окне сообщения кнопки
`MessageBoxButtons.YesNo`	**Да, Нет**
`MessageBoxButtons.YesNoCancel`	**Да, Нет, Отмена**
`MessageBoxButtons.AbortRetryIgnore`	**Прервать, Повторить, Пропустить**
`MessageBoxButtons.RetryCancel`	**Повторить, Отмена**

Параметр-иконка задает тип выводимого значка. Всего возможно отображение четырех типов иконок, задающих общий тип выводимого окна:

- информационное окно – `MessageBoxIcon.Information`;
- вопросительное окно – `MessageBoxIcon.Question`;
- окно предупреждения – `MessageBoxIcon.Warning`;
- окно с сообщением об ошибке – `MessageBoxIcon.Error`.

Определить, каким образом завершился вывод окна (то есть какая кнопка была нажата пользователем), можно, воспользовавшись данными из табл. 2.12, в которой приведены данные о возвращаемых значениях функции `MessageBox.Show`.

Таблица 2.12 ▼ Значения функции `MessageBox.Show`

Значение функции	Кнопка, с помощью которой был завершен диалог
`System.Windows.Forms.DialogResult.Ok`	**OK**
`System.Windows.Forms.DialogResult.Yes`	**Да**
`System.Windows.Forms.DialogResult.No`	**Нет**
`System.Windows.Forms.DialogResult.Cancel`	**Отмена**
`System.Windows.Forms.DialogResult.Abort`	**Прервать**
`System.Windows.Forms.DialogResult.Retry`	**Повторить**
`System.Windows.Forms.DialogResult.Ignore`	**Пропустить**

Обработка таких ошибок очень полезна. Она говорит о качестве написания программы, то есть программист подумал о том, что пользователь может ошибиться. Однако еще лучшей стратегией будет сделать так, чтобы у пользователя вообще не было возможности совершить ошибку. Как вы помните, возможная ошибка в нашей программе – это ввод исходных данных неправильного формата. Можно контролировать процесс ввода значений коэффициентов в соответствующие поля.

Напомню, что у нас три поля ввода – TextBox1, TextBox2 и TextBox3. Они работают совершенно одинаково – их задача дать возможность пользователю ввести значения коэффициентов квадратного уравнения, которые являются числами. При этом мы помним, что разделителем целой и дробной частей выступает запятая.

Для контроля ввода значений коэффициентов создайте обработчик события KeyPress и назначьте его для обработки события всех трех компонентов. Обработчик нажатия клавиш, контролирующий ввод исходных данных, приведен в листинге 2.5.

Листинг 2.5 ▾ Контроль ввода данных

```
procedure TWinForm.CheckInput(sender: System.Object;
                        e: System.Windows.Forms.KeyPressEventArgs);
begin
  // Контроль вводимых символов.
  if not (e.KeyChar in [#8,',','0'..'9'])
then
    e.Handled := true;
end;
```

Кратко поясню код. При возникновении события нажатия клавиши в обработчик передается символ, который должен быть добавлен к уже введенным. Передается он в свойстве параметра обработчика события e.KeyChar. Соответственно, нам надо проверить, входит ли этот символ в диапазон допустимых символов для ввода. Напомню, что допустимыми символами являются цифры от 0 до 9 и запятая, являющаяся разделителем целой и дробной частей. Если вы внимательно посмотрите на диапазон, представленный в условии в листинге, то увидите, что в самом начале стоит странная конструкция #8. Эта запись обозначает символ с кодом 8, то есть клавишу BackSpace. Служебное слово **not** перед оператором сравнения **in** говорит о том, что секция **then** оператора **if** будет выполняться только в том случае, если результат сравнения будет неверным, то есть введенный символ не является разрешенным. В этом случае нам надо запретить ввод этого символа.

Для того чтобы в данном случае отменить добавление введенного символа, мы присваиваем свойству Handled переданного в обработчик события значение true, которое говорит, что событие уже обработано. В этом случае выполнение обработки события прерывается и не доходит до компонента, в котором и происходит добавление нового символа к уже введенным. Таким образом, мы решили задачу контроля ввода.

Дополнительные возможности по выявлению ошибок в тексте программы

Перед тем как окончательно настроить нашу программу, сделаем небольшое отступление. Зачастую случается так, что выявить ошибку времени выполнения достаточно сложно. В нашем случае это было просто – после ввода неправильных данных возникала соответствующая ошибка. Но бывают ситуации, когда причина возникновения ошибки непонятна. В этом случае мы можем воспользоваться дополнительными возможностями среды разработки по выявлению ошибок в исходном коде. Далее я коротко остановлюсь на основных способах выявления ошибок (средствах отладки программы).

Итак, основные способы выявления ошибок:

- использование контрольных точек останова;
- просмотр списка текущих значений переменных (Watches list).

Точки останова (Breakpoints) используются в случае, когда необходимо выявить место, где происходит ошибка. Введение таких точек позволяет останавливать процесс выполнения программы и выполнять ее пошагово, от точки к точке.

Для того чтобы добавить точку останова, достаточно щелкнуть по номеру строки программы. В результате будет добавлена точка останова, которая помечается красным кружком (рис. 2.24).

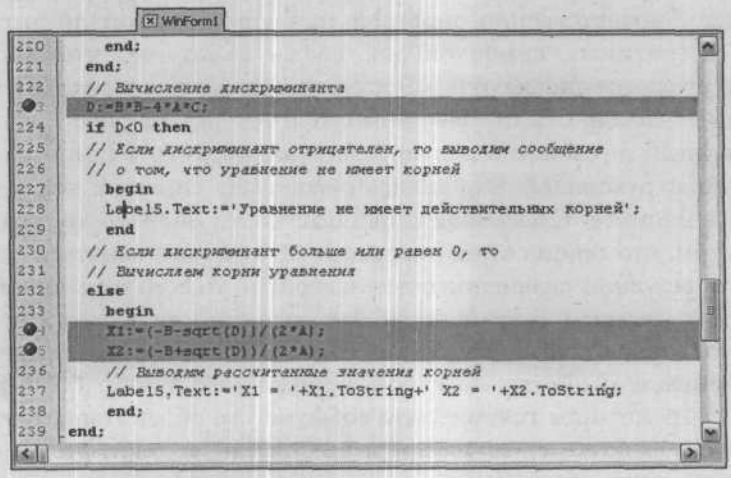

Рис. 2.24 ▼ Добавление точек останова

Аналогичного результата можно добиться, если в окне редактора кода нажать клавишу **F5** или использовать контекстное меню. Для добавления точки останова с помощью контекстного меню (рис. 2.25) в нем необходимо выбрать пункт **Debug ➤ Toggle Breakpoint**.

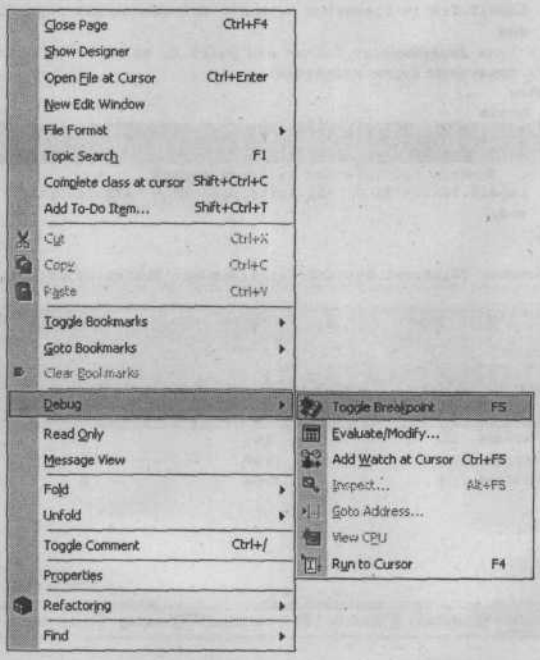

Рис. 2.25 ▾ Добавление точек остановка с помощью контекстного меню

Информацию о добавленных точках останова можно посмотреть в специальном окне (рис. 2.26), включить которое можно нажатием комбинации клавиш **Ctrl+Alt+B** или с помощью пунктов меню **View ➤ Debug Windows ➤ Breakpoints**. В окне отображаются все точки останова модуля, а также номера строк, на которых они находятся. Для быстрого перемещения к какой-либо точке достаточно выполнить на ее имени двойной щелчок мышью.

В этом же окне можно с помощью переключателей включать или выключать соответствующие точки останова, а также добавлять или удалять их.

Как же работают точки останова? После того как все точки выставлены, программа запускается обычным способом (с помощью меню **Run**). Как только программа дойдет до инструкции, напротив которой стоит точка останова,

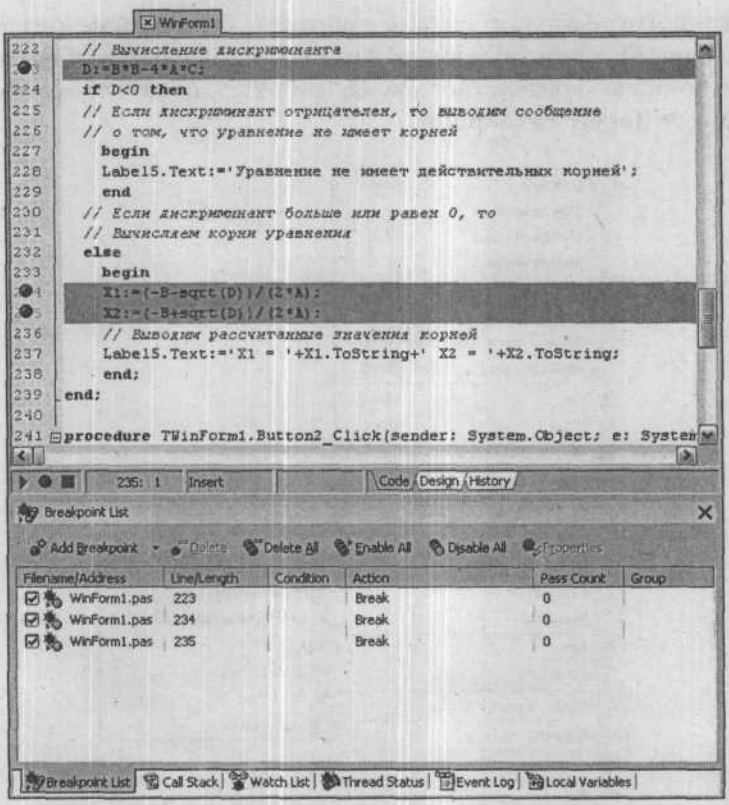

Рис. 2.26 ▼ Просмотр информации о точках останова

выполнение программы останавливается. Для дальнейшего выполнения программы следует нажать клавишу **F8** или воспользоваться пунктами **Run ➤ Step Over** главного меню. Как только вы продолжите выполнение программы одним из этих двух способов, будут выполнены все инструкции вплоть до следующей точки останова. Применение точек останова зачастую помогает понять, в каком месте возникает ошибка.

Теперь скажем несколько слов о поиске ошибок с помощью *списка просмотра значений переменных* (Watches list). Во время выполнения программы довольно часто случается так, что внешне все выглядит вроде бы нормально – программа работает, ошибок времени выполнения не возникает, но итоговый результат неправильный. Ошибка зачастую кроется в неправильных расчетах – неверно

используются переменные либо неправильно записаны вычисляемые выражения. В этом случае полезным оказывается просмотр списка значений переменных. Используется данный список, как правило, совместно с точками останова программы.

Итак, чтобы сформировать данный список, необходимо:

1. Установить курсор на переменную, значение которой мы хотим отслеживать по ходу выполнения программы.
2. Нажать комбинацию клавиш **Ctrl+F5** или же воспользоваться пунктами контекстного меню **Debug ➤ Add Watch at Cursor** (рис. 2.27).

После того как желаемые переменные добавлены, список просмотра можно посмотреть в том же окне, где мы наблюдали точки останова. Для отображения этого окна необходимо воспользоваться комбинацией клавиш **Ctrl+Alt+W** или пунктами главного меню **View ➤ Debug Windows ➤ Watches**. В результате отобразится то же самое окно, у которого будет активна вкладка **Watch List** (рис. 2.28).

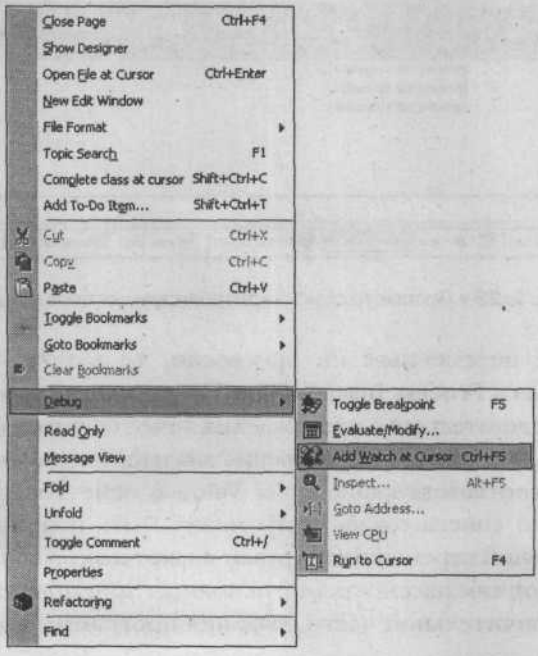

Рис. 2.27 ▾ Добавление переменной в список просмотра

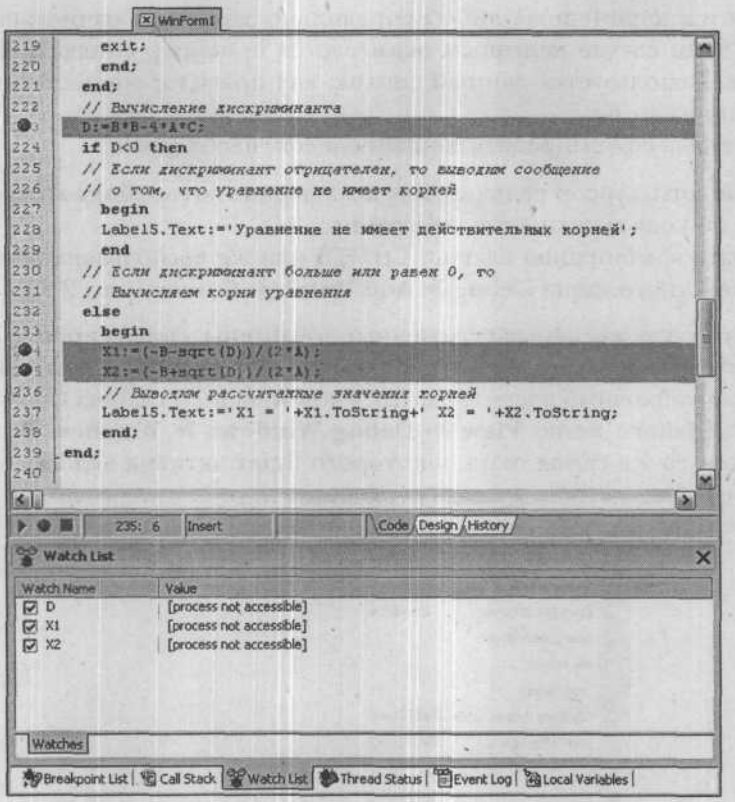

Рис. 2.28 ▼ Просмотр списка контролируемых переменных

Пока значение переменным не присвоено, то в поле **Value** (Значение) отображается надпись **Process inaccessible** (Процесс недоступен). После запуска программы и последовательного прохождения точек останова (для наглядности я выбрал точки останова в местах присвоения значений переменным) вы увидите, как заполняются соответствующие поля **Value** в окне **Watch List** (рис. 2.29). Применение такого списка очень часто может быть полезным для проверки правильности значений переменных на различных этапах выполнения программы.

На этом мы закончим рассмотрение основных приемов отладки программы и перейдем к заключительной части создания программы – ее окончательной настройке.

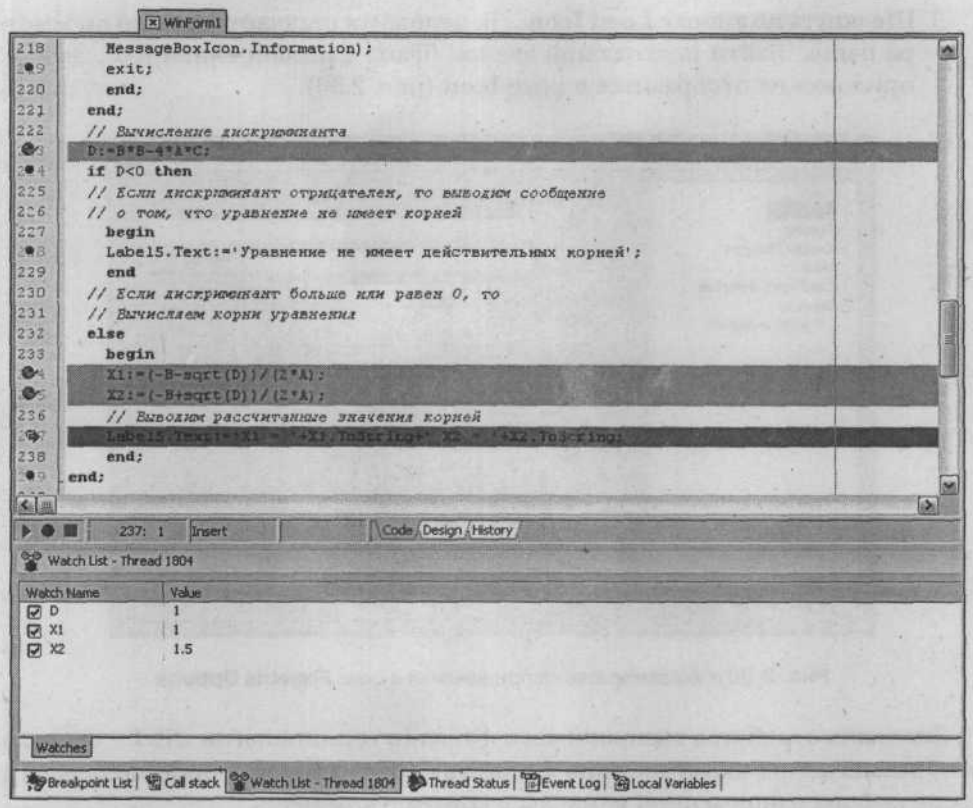

Рис. 2.29 ▾ Использование точек останова и списка просмотра значений переменных

Выполняем окончательную настройку программы

После того как программа отлажена, можно выполнить ее окончательную настройку: задать иконку (значок) приложения, а также задать атрибуты выполняемого файла (сборки).

Чтобы присвоить приложению значок, необходимо:

1. В меню **Project** выбрать команду **Options**.
2. В появившемся диалоговом окне **Project Options** выбрать раздел **Application**.

3. Щелкнуть по кнопке **Load Icon...** и, используя стандартное окно просмотра папок, найти подходящий значок (файл с расширением ico). Значок приложения отобразится в поле **Icon** (рис. 2.30).

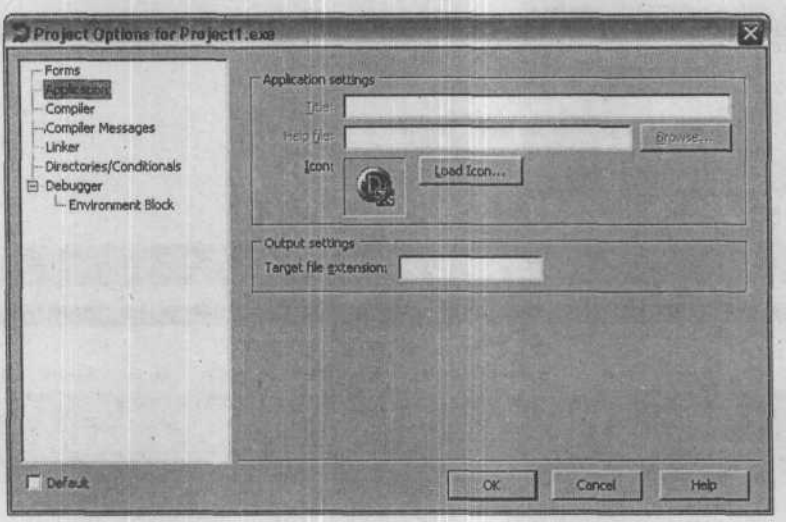

Рис. 2.30 ▼ Указание значка приложения в окне **Projects Options**

Значения атрибутов выполняемого файла (в терминологии .NET – сборки) отображаются на вкладках окна **Свойства** (рис. 2.31), которое появляется после выбора одноименной команды в контекстном меню.

Значения атрибутов можно установить в файле проекта. Для этого необходимо в меню **Project** (Проект) выбрать команду **View Source** (Просмотр). В табл. 2.13 приведено описание некоторых атрибутов выполняемого файла, а в листинге 2.6 – фрагмент кода, устанавливающий значения атрибутов программы вычисления корней квадратного уравнения.

Таблица 2.13 ▼ Атрибуты сборки

Атрибут сборки	Описание	Где отображается атрибут
`AssemblyTitle`	Название программы	В строке **Описание** вкладки **Общие** окна **Свойства**
`AssemblyDescription`	Краткое описание программы	В поле **Значение** вкладки **Версия** окна **Свойства**
`AssemblyCopyright`	Информация об авторских правах	В строке **Авторские права** вкладки **Версия** окна **Свойства**

Таблица 2.13 ▾ Атрибуты сборки (окончание)

Атрибут сборки	Описание	Где отображается атрибут
`AssemblyProduct`	Информация о продукте	В группе элементов **Имя элемента** при выборе пункта **Название продукта** вкладки **Версия** окна **Свойства**
`AssemblyVersion`	Информация о версии продукта	Там же, при выборе пункта **Версия продукта**
`AssemblyCompany`	Информация о производителе	Там же, при выборе пункта **Производитель**

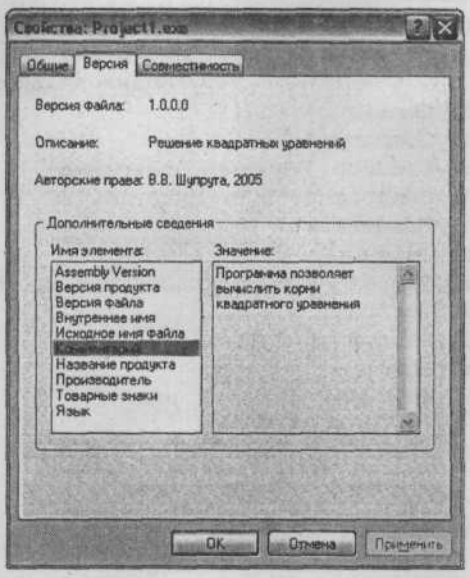

Рис. 2.31 ▾ Значение атрибутов сборки в окне **Свойства**

Листинг 2.6 ▾ Задание значений атрибутов в файле проекта

```
program Project1;
 {%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'}
 {$R 'WinForm1.TWinForm1.resources' 'WinForm1.resx'}
uses
  System.Reflection,
  System.Runtime.CompilerServices,
  System.Windows.Forms,
  WinForm1 in 'WinForm1.pas' {WinForm1.TWinForm1:
                                        System.Windows.Forms.Form};
{$R *.res}
{$REGION 'Program/Assembly Information'}
    [assembly: AssemblyDescription('Программа позволяет вычислить'+
                                   'корни квадратного уравнения')]
[assembly: AssemblyConfiguration('')]
[assembly: AssemblyCompany('')]
[assembly: AssemblyProduct('Учебная программа')]
[assembly: AssemblyCopyright('B.B. Шупрута, 2005')]
[assembly: AssemblyTrademark('')]
[assembly: AssemblyCulture('')]
[assembly: AssemblyTitle('Решение квадратных уравнений')]
[assembly: AssemblyVersion('1.0')]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile('')]
[assembly: AssemblyKeyName('')]
{$ENDREGION}
[STAThread]
begin
  Application.Run(TWinForm1.Create);
end.
```

На этом разработка программы (проекта для .NET) закончена. Следующая часть главы посвящена разработке проектов для платформы Win32.

Создаем первый проект для Win32

В начале этой главы мы подробно рассмотрели процесс написания программы для .NET, рассчитывающей корни квадратного уравнения. Теперь создадим аналогичную программу для платформы Win32. Я не буду слишком подробно описывать технологию создания проекта – принцип действий и их

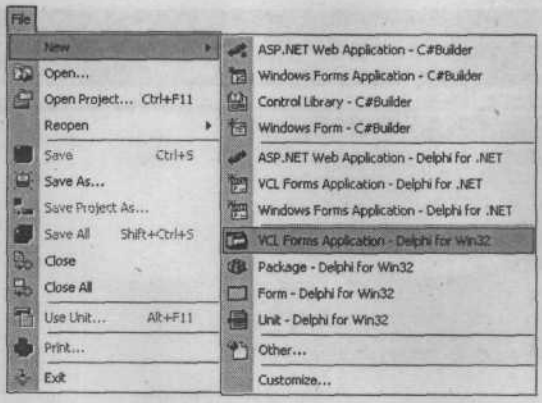

Рис. 2.32 ▾ Создание проекта типа **VCL Forms Application**

последовательность работает и здесь. Внимание будет уделяться в основном отличительным особенностям при создании проекта.

Чтобы начать работу в Borland Delphi 2005 над проектом данного типа, необходимо в меню **File** выбрать команду **New**, и в появившемся списке выбрать пункт **VCL Forms Application – Delphi for Win32** (рис. 2.32).

Настраиваем будущее окно программы

После создания нового проекта в центральной части экрана будет окно дизайнера формы (рис. 2.33).

Как, видите, внешний вид формы несколько изменился. Кроме того, если обратить внимание на окно **Object Inspector**, то сразу будут заметно, что и свойства формы изменились. Начнем работу над новым проектом с настройки формы.

Первое отличие, которое мы наблюдаем, – у формы нет свойства `Text`. Теперь за заголовок формы отвечает свойство `Caption`. Попробуйте самостоятельно изменить значение свойства `Caption` – щелкните мышью по строке **Caption** и введите текст «Вычисление корней квадратного уравнения», затем нажмите клавишу **Enter**.

Свойства, определяющие размер формы, также изменились. Теперь размер формы определяется значениями свойств `Width` и `Height`, которые соответственно задают размеры формы по горизонтали и вертикали. Чтобы не заострять внимание на всех отличиях, мы отразили основные свойства формы в табл. 2.14.

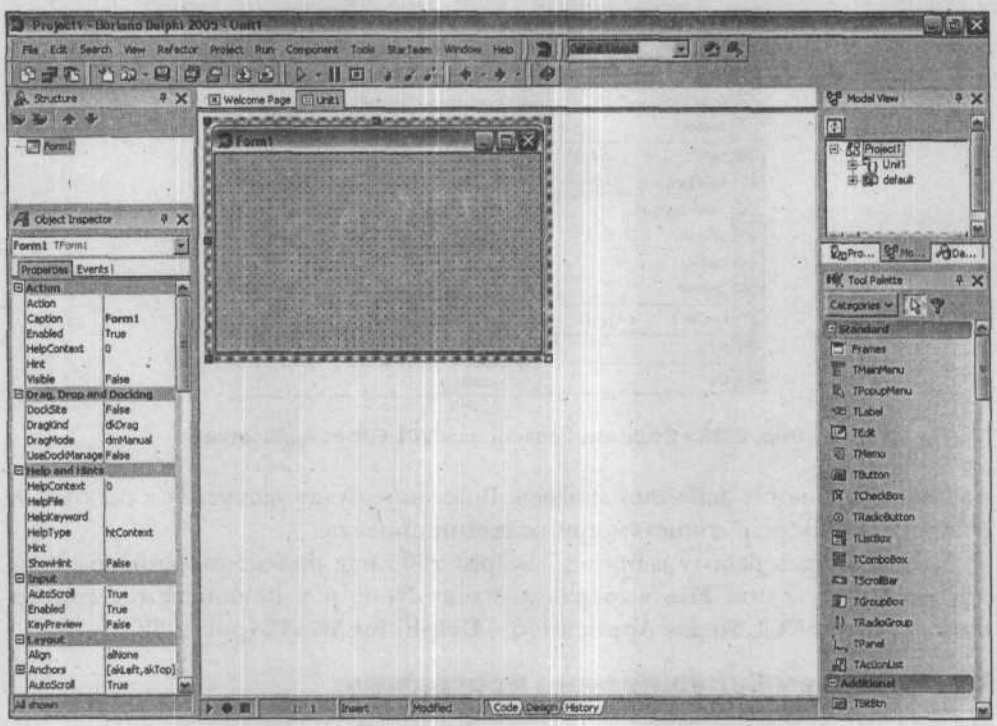

Рис. 2.33 ▼ Начало работы над проектом

Таблица 2.14 ▼ Основные свойства формы

Свойство	Значение	Комментарий
`Caption`	Вычисление корней квадратного уравнения	Заголовок формы
`Width`	392	Размер формы в пикселях по горизонтали
`Height`	184	Размер формы в пикселях по вертикали
`Font.Name`	Microsoft Sans Serif	Наименование используемого шрифта (для компонентов, которые будут помещены на форму)
`Font.Size`	8	Размер используемого шрифта
`BorderStyle`	`bsSingle`	Свойство определяет доступность изменения размеров окна программы во время выполнения. Значение свойства **bsSingle** запрещает такое изменение

Таблица 2.14 ▾ Основные свойства формы (окончание)

Свойство	Значение	Комментарий
`Position`	`poScreenCenter`	Задает место появления окна программы на экране, `poScreenCenter` – в центре экрана
`biMaximize`	`False`	Доступность кнопки **Развернуть** в правом верхнем углу программы. `False` – кнопка недоступна

После задания основных свойств формы ваше будущее окно должно приобрести вид, показанный на рис. 2.34.

Рис. 2.34 ▾ Вид формы после изменения ее основных свойств

Придаем программе необходимый внешний вид

Форма настроена; теперь самое время взглянуть на палитру компонентов – окно **Tool Palette**. Если внимательно посмотрим на это окно (рис. 2.35), то увидим, что название и состав компонентов изменились: теперь используется другая библиотека компонентов, специально предназначенная для платформы Win32 (библиотека имеет аббревиатуру VCL – Visual Components Library, библиотека визуальных компонентов).

Не стоит пугаться новых названий – хотя это и другие компоненты, с их помощью мы также можем настраивать форму: добавлять поля для ввода, метки и кнопки. Отличия при использовании новых компонентов заметны в основном в названиях свойств и методов. Ниже приведена табл. 2.15, в которой можно увидеть, с помощью каких компонентов мы создавали прошлый проект, а также их аналоги в Win32, которыми мы воспользуемся сейчас.

Рис. 2.35 ▼ Окно **Tool Palette** теперь содержит другой набор компонентов

Таблица 2.15 ▼ Компоненты, используемые для создания программы решения квадратного уравнения

Назначение компонента	Проект для .NET	Проект для Win32
Ввод исходных данных	`TextBox`	`TEdit`
Отображение текстовых данных	`Label`	`TLabel`
Командная кнопка	`Button`	`TButton`

Теперь вернемся к нашей программе. Из табл. 2.15 мы выяснили, что для настройки формы нам необходимо воспользоваться компонентами `TEdit`, `TLabel`, и `TButton`. Ниже в табл. 2.16–2.21 приведены основные свойства этих компонентов, а также значения этих свойств, которые необходимо изменить.

Таблица 2.16 ▾ Основные свойства компонента `TEdit`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Text`	Текст, который содержится в поле для редактирования
`Font`	Шрифт, который используется для отображения текста
`Color`	Цвет фона поля компонента
`Left`	Расстояние от левой границы формы до левой границы компонента
`Top`	Расстояние от верхней границы формы до верхней границы компонента
`Width`	Ширина поля компонента
`Height`	Высота поля компонента
`BorderStyle`	Вид рамки компонента. По умолчанию задается обычная (`bsSingle`) рамка. Свойство также может принимать значение `None` (рамка отсутствует)

В табл. 2.17 приведены значения свойств компонентов `TEdit1`, `TEdit2` и `TEdit3`, которые нам необходимо изменить. Значения остальных свойств мы менять не будем, и поэтому они в таблице не рассмотрены.

Таблица 2.17 ▾ Значения свойств компонентов `TEdit1-TEdit3`

Свойство	TEdit1	TEdit2	TEdit3
`Text`			
`Left`	40	40	40
`Top`	48	76	104
`Width`	48	48	48

После настройки компонентов для ввода данных перенесите на форму нужное количество компонентов для отображения данных (`TLabel1-TLabel5`). Свойства этого компонента, которые представляют для нас интерес при написании этой программы, приведены в табл. 2.18, значения свойств компонентов `TLabel1-TLabel5` – в табл. 2.19.

Таблица 2.18 ▾ Свойства компонента `TLabel`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Caption`	Отображаемый в поле компонента текст
`Font`	Шрифт, который используется для отображения текста

Таблица 2.18 ▼ Свойства компонента `TLabel` (окончание)

Свойство	Комментарий
`Color`	Цвет фона поля компонента
`Left`	Расстояние от левой границы формы до левой границы компонента
`Top`	Расстояние от верхней границы формы до верхней границы компонента
`Width`	Ширина поля компонента
`Height`	Высота поля компонента
`Alignment`	Способ выравнивания текста в поле компонента. Текст может быть прижат к левому краю (`taLeft`), правому краю (`taRight`) или быть выровненным по центру (`taCenter`)

Таблица 2.19 ▼ Свойства компонентов `TLabel1-TLabel5`

Свойство	TLabel1	TLabel2	TLabel3	TLabel4	TLabel5
`Caption`	Введите значения коэффициентов квадратного уравнения	A:	B:	C:	
`Left`	24	24	24	24	104
`Top`	16	55	83	111	48
`Width`	328	16	16	16	120
`Height`	24	16	16	16	80

Последнее, что осталось сделать, это добавить на форму и настроить командные кнопки – два компонента типа `TButton`. Свойства этого компонента приведены в табл. 2.20.

Таблица 2.20 ▼ Основные свойства компонента `TButton`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Caption`	Отображаемый на кнопке текст
`Font`	Шрифт, используемый для отображения текста на кнопке
`Left`	Расстояние от левой границы формы до левой границы компонента
`Top`	Расстояние от верхней границы формы до верхней границы компонента
`Width`	Ширина кнопки
`Height`	Высота кнопки
`Enabled`	Признак доступности кнопки. Если значение свойства равно `True`, то кнопка доступна, если `False` – кнопка недоступна

Установив значения свойств компонентов TButton1 и TButton2 в соответствии с данными табл. 2.21, мы должны получить форму, вид которой показан на рис. 2.36.

Таблица 2.21 ▾ Значения свойств компонентов TButton1 и TButton2

Свойство	TButton1	TButton2
Caption	Вычислить	Завершение работы
Left	240	240
Top	48	96
Width	120	120
Height	32	32

Рис. 2.36 ▾ Окончательный вид формы после настройки всех компонентов

Работа над формой завершена. Теперь можно приступать к написанию текста программы – процедур обработки события Click на командных кнопках Button1 и Button2.

Процедуры обработки событий создаются уже известным нам способом – с использованием окна **Object Inspector**. После выбора кнопки Button1 (кнопка **Вычислить**) двойной щелчок напротив надписи Click на вкладке Events этого окна создает заготовку для процедуры обработки события Click – TForm1.Button1Click. После того как заготовка для процедуры будет создана, необходимо написать исходный код, подобный тому, что был приведен в листинге 2.1.

В листинге 2.7 приведен текст функции обработки события Click на кнопке **Вычислить** (Button1).

Листинг 2.7 ▾ Процедура обработки события Click

```
procedure TForm1.Button1Click(Sender: TObject);
var A,B,C:single;
```

```
    D:real;
    X1,X2:real;
begin
  // Ввод исходных данных.
  A:=StrToFloat(Edit1.Text);
  B:= StrToFloat(Edit2.Text);
  C:= StrToFloat(Edit3.Text);
  // Вычисление дискриминанта.
  D:=B*B-4*A*C;
  if D<0 then
    // Если дискриминант отрицателен, то выводим сообщение
    // о том, что уравнение не имеет корней.
    begin
    Label5.Caption:='Уравнение не имеет действительных корней';
    end
    // Если дискриминант больше или равен 0, то
    // вычисляем корни уравнения.
    else
      begin
      X1:=(-B-sqrt(D))/(2*A);
      X2:=(-B+sqrt(D))/(2*A);
      // Выводим рассчитанные значения корней.
      Label5.Caption:='X1 = '+FloatToStr(X1)+
      Chr(13)+ // Перевод на следующую строку.
      'X2 = '+FloatToStr(X2);
      end;
end;
```

Как видно из листинга 2.7, структура программы не претерпела существенных изменений, тем не менее синтаксис отдельных элементов изменился. Разберем приведенный в листинге исходный код.

Функция обработки `TForm1.Button1Click` выполняет вычисление корней квадратного уравнения и выводит результат в метку `Label5`. Исходные данные о коэффициентах при неизвестном вводятся из полей редактирования `Edit1`, `Edit2` и `Edit3`. После ввода данных пользователем значение свойств `Text` этих компонентов меняется на те, что ввел пользователь. Для преобразования введенных пользователем строк в числа используется функция `StrToFloat`. В качестве параметра этой функции передается текстовое изображение числа. Результатом этой функции является число, полученное из введенной пользователем строки. Поэтому если пользователь введет какой-либо текст, который невозможно перевести в число, то будет выдано сообщение об ошибке (функция работает аналогично рассмотренной нами ранее функции `ToSingle`

при разработке проекта для .NET). Таким образом, функция `StrToFloat` сработает только в том случае, если передаваемое число записано в верном формате.

Вычисленные значения корней выводятся в поле `Label5` путем присваивания значения свойству `Caption`. Для обратного преобразования числа в строку используется обратная функция `FloatToStr`.

Изучаем структуру проекта для Win32

Теперь разберем особенности структуры созданного проекта. Далее приведен список файлов, который формируется при создании любого проекта под Win32 (после выполнения операции сохранения). Реально проект в простейшем случае представляет собой совокупность следующих файлов:

- файл описания проекта (bdsproj-файл);
- главный модуль (dpr-файл);
- файл ресурсов (res-файл);
- файл конфигурации (cfg-файл);
- модуль формы (dfm-файл);
- модуль реализации (pas-файл).

Файл описания проекта

Файл описания проекта (bdsproj-файл) имеет ту же функцию, что и в проектах .NET. Он представляет собой файл специального формата, в котором записана общая информация о проекте. Изменять этот файл вручную также не рекомендуется.

Главный модуль

В главном модуле (dpr-файле) содержатся инструкции, обеспечивающие запуск нашей программы. В качестве примера в листинге 2.8 приведен главный текст файла главного модуля нашей программы (файл будет создан, как только мы выполним сохранение проекта).

Листинг 2.8 ▾ Главный модуль программы

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
  Application.Initialize;
```

```
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

В заголовке модуля указано имя программы, далее следуют команды, которые содержат информацию о том, что именно (какие модули) понадобится компилятору Delphi для создания конечного exe-файла. Как видите, в отличие от .NET-проектов здесь не требуется наличия специальных библиотек – вся информация будет храниться в формируемом exe-файле.

Модуль формы

Модуль формы содержит только информацию о ее настройках и компонентах, которые на ней присутствуют. Модуль формы формируется автоматически при выполнении настроек формы, перенесении на нее и настройки компонентов. А где же хранятся процедуры обработки событий для этих компонентов? Процедуры обработки событий хранятся в файле реализации (см. ниже). В листинге 2.9 приведен текст модуля формы нашей программы.

Листинг 2.9 ▼ Модуль формы программы

```
object Form1: TForm1
  Left = 0
  Top = 0
  BorderIcons = [biSystemMenu, biMinimize]
  BorderStyle = bsSingle
   Caption = #1042#1099#1095#1080#1089#1083#1077#1085#1080#1077'
'#1082#1086#1088#1085#1077#1081'
'#1082#1074#1072#1076#1088#1072#1090#1085#1086#1075#1086'
'#1091#1088#1072#1074#1085#1077#1085#1080#1103
  ClientHeight = 152
  ClientWidth = 386
  Color = clBtnFace
  Font.Charset = RUSSIAN_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  Position = poScreenCenter
  PixelsPerInch = 96
  TextHeight = 13
  object Label1: TLabel
```

```
    Left = 24
    Top = 16
    Width = 299
    Height = 13
    Caption = #1042#1074#1077#1076#1080#1090#1077'
'#1079#1085#1072#1095#1077#1085#1080#1103'
'#1082#1086#1101#1092#1092#1080#1094#1080#1077#1085#1090#1086#1074'
'#1082#1074#1072#1076#1088#1072#1090#1085#1086#1075#1086'
'#1091#1088#1072#1074#1085#1077#1085#1080#1103
  end
  object Label2: TLabel
    Left = 24
    Top = 55
    Width = 10
    Height = 13
    Caption = 'A:'
  end
  object Label3: TLabel
    Left = 24
    Top = 83
    Width = 10
    Height = 13
    Caption = 'B:'
  end
  object Label4: TLabel
    Left = 24
    Top = 111
    Width = 10
    Height = 13
    Caption = 'C:'
  end
  object Label5: TLabel
    Left = 104
    Top = 48
    Width = 120
    Height = 80
    AutoSize = False
    WordWrap = True
  end
  object Edit1: TEdit
    Left = 40
    Top = 48
    Width = 48
```

```
    Height = 21
    TabOrder = 0
  end
  object Edit2: TEdit
    Left = 40
    Top = 76
    Width = 48
    Height = 21
    TabOrder = 1
  end
  object Edit3: TEdit
    Left = 40
    Top = 104
    Width = 48
    Height = 21
    TabOrder = 2
  end
  object Button1: TButton
    Left = 240
    Top = 48
    Width = 120
    Height = 32
    Caption = #1042#1099#1095#1080#1089#1083#1080#1090#1100
    TabOrder = 3
    OnClick = Button1Click
  end
  object Button2: TButton
    Left = 240
    Top = 96
    Width = 120
    Height = 32
     Caption = #1047#1072#1074#1077#1088#1096#1077#1085#1080#1077'
'#1088#1072#1073#1086#1090#1099
    TabOrder = 4
  end
end
```

Модуль реализации

В модуле реализации содержится информация только о присутствующих на форме компонентах и процедурах обработки событий на этих компонентах.

Модуль формы можно условно разделить на две части – секцию описания и собственно раздел реализации.

Секция описания (начинается с служебного слова `interface`) содержит автоматически сформированное Delphi-объявление типа формы. Этот раздел также содержит перечень компонентов, которые находятся на форме.

Секция реализации начинается со служебного слова `implementation`. Она содержит объявление локальных процедур и функций, в том числе и процедур обработки событий.

Как и в случае проекта для .Net, могут существовать еще и секции инициализации и финализации. Но в нашем случае они не используются.

Теперь необходимо отметить некоторые особенности проекта для Win32.

Посмотрите на текст модуля реализации проекта для Win32 (листинг 2.10).

Листинг 2.10 ▾ Текст модуля реализации

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

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

```
var A,B,C:single;
    D:real;
    X1,X2:real;
begin
  // Ввод исходных данных.
  A:=StrToInt(Edit1.Text);
  B:=StrToInt(Edit2.Text);
  C:=StrToInt(Edit3.Text);
  // Вычисление дискриминанта.
  D:=B*B-4*A*C;
  if D<0 then
    // Если дискриминант отрицателен, то выводим сообщение
    // о том, что уравнение не имеет корней.
    begin
    Label5.Caption:='Уравнение не имеет действительных корней';
    end
    // Если дискриминант больше или равен 0, то
    // вычисляем корни уравнения.
    else
      begin
      X1:=(-B-sqrt(D))/(2*A);
      X2:=(-B+sqrt(D))/(2*A);
      // Выводим рассчитанные значения корней.
      Label5.Caption:='X1 = '+FloatToStr(X1)+
      Chr(13)+ // Перевод на следующую строку.
      'X2 = '+FloatToStr(X2);
      end;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
end.
```

В модуле формы проекта для .NET присутствовало все – и описание компонентов (значения их свойств), и соответствующие им процедуры обработки событий. В проектах для Win32 эта информация разделена на модуль формы и модуль реализации. Модуль реализации содержит перечень (условно говоря, ссылки) компонентов и соответствующие им процедуры обработки. Каким же образом компилятор узнает описание компонентов в проекте Win32, на что осуществляется ссылка в модуле реализации? Ответ прост – посмотрите внимательно на начало секции **implementation**. Сразу в начале этого раздела

вы увидите директиву (не путайте с комментариями) {$R *.dfm}. Данная директива означает, что описание компонентов (значения их свойств) находятся в файле unit1.pas. Звездочка (*) в данном случае означает, что имя файла модуля формы совпадает с именем файла модуля реализации.

Преобразуем исходный текст программы в исполняемый файл

После сохранения проекта приступайте к компиляции с помощью пункта **Compile** меню **Project**. Результатом компиляции будет exe-файл, совпадающий с именем проекта. Если вы сравните размеры файлов программ для .NET и Win32, то заметите, что первый занимает всего 30 Кб, второй – почти 400 Кб. Почему такая большая разница, ведь по сути программы делают идентичные действия и даже внешне состоят практически из одних и тех же компонентов?

Такое существенное различие в размере выходного файла объясняется просто. При компиляции в .NET в директивах главного модуля были ссылки на библиотеки из Microsoft .NET Framework. *Ссылки*, но не сами библиотеки. Исполняемый файл формировался с учетом того, что вся необходимая информация (информация о форме, компонентах и используемых функциях) хранилась во *внешних* библиотеках. Поэтому его размер достаточно мал. Достаточно большой размер файла, полученного компилятором для Win32, объясняется тем, что вся необходимая информация собрана *внутри* файла.

Соответственно, для работы первого проекта необходимо наличие установленного пакета Microsoft .NET Framework (библиотек, указанных в директивах главного модуля проекта). Во втором случае этого не требуется, полученный файл абсолютно автономен.

Запускаем полученную программу

Запустите программу любым удобным для вас способом (из среды разработки или Проводника). После некоторого количества выполненных расчетов мы заметим, что в случае если корни уравнения являются дробными числами, то результат отображается не очень красиво (рис. 2.37).

Для более красивого вывода можно использовать другой вариант отображения чисел – функцию `FloatToStrF`, которая позволяет программисту самому определить внешний вид отображаемого числа (тип отображения, общее количество знаков, количество знаков после запятой). В общем случае форма использования функции `FloatToStrF` выглядит следующим образом:

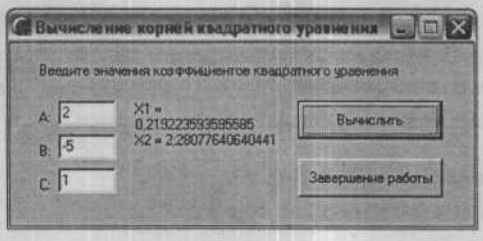

Рис. 2.37 ▼ Дробные значения по умолчанию вычисляются с большой точностью

```
FloatToStrF(<Number>,<Type>,<precision>,<digits>)
```

Параметр `<Number>` содержит переводимое в строку число, `<Type>` определяет тип отображения, `<precision>` – точность, `<digits>` – количество цифр после запятой.

Тип отображения по аналогии с рассмотренными ранее в табл. 2.9 может принимать следующие значения:

- универсальный (основной) формат – `ffGeneral`;
- научный формат (с плавающей точкой) – `ffExponent`;
- число с фиксированной точкой – `ffFixed`;
- числовой формат – `ffNumber`;
- денежный формат – `ffCurrency`.

В листинге 2.11 приведен фрагмент (с использованием функции `FloatToStrF`) исходного кода процедуры обработки события `Click` на кнопке `Button1`.

Листинг 2.11 ▼ Использование функции `FloatToStrF`

```
// Выводим рассчитанные значения корней.
Label5.Caption:='X1 = '+FloatToStrF(X1,ffFixed,5,2)+
 Chr(13)+ // Перевод на следующую строку.
 'X2 = '+FloatToStrF(X2,ffFixed,5,2);
```

Результатом использования подобного кода будет более красивый вывод результата (рис. 2.38).

Последнее, чему следует уделить внимание, – это обработка возможных ошибок в процессе выполнения программы. Как мы уже знаем, в нашей программе есть возможность ввода неправильных данных со стороны пользователя, поэтому в тексте программы это необходимо предусмотреть. Сделать это можно с помощью уже знакомой нам инструкции **`try`** (см. листинг 2.12).

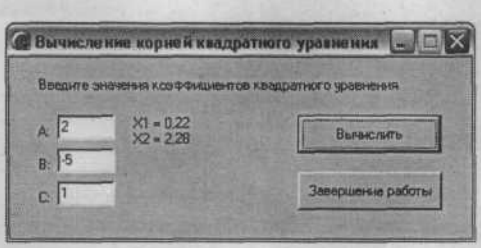

Рис. 2.38 ▾ Результат использования функции `FloatToStrF`

Листинг 2.12 ▾ Итоговый вариант процедуры обработки события `Click` на кнопке `Button1`

```
procedure TForm1.Button1Click(Sender: TObject);
var A,B,C:single;
    D:real;
    X1,X2:real;
begin
  // Ввод исходных данных.
  try
  A:=StrToInt(Edit1.Text);
  B:=StrToInt(Edit2.Text);
  C:=StrToInt(Edit3.Text);
  except
  On EConvertError do
    begin
      if (Edit1.Text='') or (Edit2.Text='') or (Edit3.Text='')
        then
        MessageDlg('Необходимо ввести все коэффициенты',
                                          mtInformation,[mbOK],0)
        else
        MessageDlg('При задании коэффициентов используйте числа',
                                          mtInformation,[mbOK],0);
      Exit;
    end;
  end;
  // Вычисление дискриминанта.
  D:=B*B-4*A*C;
  if D<0 then
    // Если дискриминант отрицателен, то выводим сообщение
    // о том, что уравнение не имеет корней.
    begin
    Label5.Caption:='Уравнение не имеет действительных корней';
    end
```

```
    // Если дискриминант больше или равен 0, то
    // вычисляем корни уравнения.
    else
      begin
      X1:=(-B-sqrt(D))/(2*A);
      X2:=(-B+sqrt(D))/(2*A);
      // Выводим рассчитанные значения корней.
      Label5.Caption:='X1 = '+FloatToStrF(X1,ffFixed,5,2)+Chr(13)
                          + 'X2 = '+FloatToStrF(X2,ffFixed,5,2);
      end;
end;
```

В данном примере для предотвращения ошибок используется инструкция **try...except**. Однако в синтаксисе записи возможных вариантов исключений произошли изменения. В табл. 2.22 приведены наиболее часто возникающие исключения (аналог исключений, описанных ранее в проекте для .NET, см. в табл. 2.10).

Таблица 2.22 ▼ Типовые исключения

Тип исключения	Причина возникновения
EConvertError	Ошибка преобразования. Возникает при выполнении преобразования, если преобразуемая величина не может быть приведена к требуемому виду. Наиболее часты случаи при переводе чисел в строки и обратно
EZeroDivide	Ошибка переполнения. Возникает в результате выполнения операции при выходе за границы допустимого диапазона значения. Также возникает в случае операции деления, если делитель равен нулю
EFileError	Ошибка открытия файла (файл не найден). Возникает при выполнении файловых операций в том случае, если не удается найти необходимый файл на носителе

Для информирования пользователя о неправильно введенных данных используется функция MessageDlg. В общем случае инструкция использования этой функции выглядит следующим образом:

```
MessageDlg(<Текст>,<Тип окна>,[<Кнопки>],<Контекст справки>)
```

Параметр <Текст> задает выводимую строку текста.

С помощью параметра <Тип окна> задается тип выводимого сообщения. Сообщение может быть следующих типов:

- информационное – mtInformation;
- предупредительное – mtWarning;
- сообщение об ошибке – mtError;
- запрос на подтверждение – mtConfirmation;
- обычное – mtCustom.

Параметр <Кнопки> задает тип отображаемой в окне сообщения кнопки (допускается указание нескольких кнопок через запятую). В табл. 2.23 приведены возможные значения этого параметра.

Таблица 2.23 ▾ Константы, задающие тип кнопки в окне сообщения

Константа, определяющая тип кнопки	Выводимые в окне сообщения кнопки
mbYes	Да
mbNo	Нет
mbOk	Ok
mbCancel	Отмена
mbAbort	Прервать
mbRetry	Повторить
mbIgnore	Пропустить
mbHelp	Помощь

Параметр <Контекст справки> определяет номер окна справочной системы, которое будет выведено на экран в случае нажатия клавиши **F1**. Если использование справочной системы не предусмотрено, то значение параметра равно нулю.

Как и в предыдущем случае, мы можем контролировать символы, вводимые пользователем. Надо точно так же создать обработчик события нажатия клавиши (теперь оно называется OnKeyPress) и назначить его всем трем компонентам для ввода данных. Исходный код такого обработчика смотрите в листинге 2.13.

Листинг 2.13 ▾ Контроль ввода данных

```
procedure TForm1.CheckInput(Sender: TObject; var Key: Char);
begin
  if not (Key in [#8,',','0'..'9']) then
    Abort;
end;
```

Выглядит очень похоже, не правда ли? Единственное отличие только в том, что вместо присваивания параметру обработчика какого-либо логического значения вызывается функция Abort. Это вызов тихого исключения (Silent exception) – специальное исключение, которое не приводит к выводу сообщения об ошибке, однако успешно выполняет свою функцию – прерывает обработку события.

Выполняем окончательную настройку приложения

На заключительном этапе приложению можно задать иконку (значок), а также заголовок приложения.

Чтобы присвоить приложению значок, необходимо:

1. В меню **Project** выбрать команду **Options**.
2. В появившемся диалоговом окне **Project Options** выбрать раздел **Application**.
3. Щелкнуть по кнопке **Load Icon...** и, используя стандартное окно просмотра папок, найти подходящий значок (файл с расширением ico).

В этом же окне в поле **Title** можно указать специфический заголовок приложения.

Глава 3 Язык программирования Delphi

После некоторой практики мы снова возвращаемся к теории. В прошлой главе мы получили некоторые познания о том, как создаются программы в Borland Delphi 2005. В этой главе мы познакомимся с *основами* языка Delphi.

Зная основы языка, уже можно не только интуитивно понимать тексты программ, но и разрабатывать их самому. Забегая немного вперед, скажу, что прежде чем писать программы, мы познакомимся с их основными элементами.

Под *элементами* программы мы будем понимать минимальные неделимые ее части, которые имеют определенное значение для компилятора.

Любая программа в Delphi состоит из зарезервированных слов языка (операторов языка), идентификаторов (констант и переменных), выражений, подпрограмм (процедур и функций) и комментариев. Сложно? Конечно, сложно. Именно поэтому в предыдущей главе мы сделали свой первый проект, пока еще не до конца осознавая, что есть что в тексте программы. Теперь же нам будет гораздо проще изучать язык Delphi, имея некоторую практику за плечами.

Итак, по окончании прочтения этой главы вы уже будете представлять себе, что такое язык Delphi и будете готовы писать простейшие программы.

Но прежде всего нам необходимо изучить так называемый *алфавит языка* Delphi, что позволит понять, какие же символы можно использовать при написании программ, а какие – нет.

Изучаем алфавит языка

Алфавит языка представляет собой набор символов, которые мы можем использовать при написании программ. Алфавит языка Delphi включает в себя буквы, цифры, специальные символы, пробелы и зарезервированные слова.

Для составления кода программы можно использовать только латинские буквы от a до z (A до Z). При этом в языке Delphi нет различия между строчными и прописными буквами (если только они не входят в символьные и строковые выражения). Дополнительно с буквами может использоваться и знак подчеркивания (_). Буквы кириллицы допускается использовать только для символьных и строковых выражений.

Цифры – только арабские от 0 до 9.

К *специальным символам* относятся следующие символы:

+ – * / = . , : ; ' ^ < > ( ) { } [ ] @ # $.

Кроме специальных символов, допускается использование и специальных пар символов:

// <> >= <= := (* *) (. .)

В тексте программы такие парные символы нельзя разделять пробелами.

Особо необходимо отметить использование *пробелов*. Символ пробела используется в языке Delphi как ограничитель между зарезервированными словами, идентификаторами, константами. При отделении подобных элементов программы друг от друга допускается использование практически произвольного числа пробелов. При этом все пробелы между словами считаются *одним* пробельным символом.

Зарезервированные слова играют важную роль в Delphi, придавая программе в целом свойство текста, написанного почти на английском языке. Каждое зарезервированное слово (их в языке Delphi несколько десятков) несет в себе условное сообщение для компилятора, который анализирует текст программы так же, как читаем его мы: слева направо и сверху вниз. Зарезервированные слова в окне редактора кода обозначаются полужирным шрифтом. Подобное выделение означает, что программист не может использовать зарезервированное слово для своих целей, например создать переменную или константу с таким именем.

В табл. 3.1 приведен перечень используемых в языке Delphi зарезервированных слов.

Таблица 3.1 ▼ Основные зарезервированные слова языка Delphi

and	function	property
array	goto	raise
as	if	record

Таблица 3.1 ▾ Основные зарезервированные слова языка Delphi (окончание)

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

Назначение зарезервированных слов можно пояснить на примере. Например, пара **begin** и **end** используется для логического объединения последовательности команд, соответственно обозначая начало и конец этой последовательности.

Зарезервированные слова не могут быть использованы в качестве идентификаторов. Также не рекомендуется использовать в качестве идентификаторов стандартные директивы, приведенные в табл. 3.2.

Таблица 3.2 ▾ Стандартные директивы языка Delphi

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

Для чего нужны комментарии

Комментарии используются в программе для облегчения восприятия текста программы. В самом деле, на тот случай, если вы забудете, что означает та или иная часть программы, константа, переменная и т.д., – вас всегда выручит применение комментариев. Кроме того, во время отладки вы можете «закомментировать» любой фрагмент программы, и он не будет учитываться во время компиляции. Соответственно, когда такой код может снова понадобиться, ничто не мешает его «раскомментировать» – вы сэкономите время, так как вам не придется набирать этот код заново.

Итак, комментариями в программе называются элементы программы, не имеющие значения для компилятора. Во время преобразования исходного текста программы компилятор просто пропускает их. Отличительной особенностью комментариев является то, что они помечаются курсивом в окне редактора исходного кода.

Существует два типа комментариев. Первый тип – *однострочные комментарии*. Для того чтобы сделать однострочный комментарий, поставьте два символа //. В итоге компилятор будет считать, что все символы, начиная с // и до конца этой строки, являются комментарием:

```
...
// Эта строка является комментарием.
...
a := 5;  // Это тоже комментарий, но оператор присваивания
         // будет скомпилирован.
...
```

Второй тип комментария – *многострочный комментарий*. Delphi поддерживает как старую модель многострочного комментария, унаследованную от Паскаля, так и новую. Старая, паскалевская, модель начинается с комбинации символов (* и заканчивается *). Все, что находится между этими парами символов, рассматривается компилятором как комментарий. Новая модель формируется с помощью открывающей и закрывающей фигурной скобки, но работает точно так же, как и старая:

```
...
(*
Это комментарий
в стиле Паскаля
```

```
*)
...
{
эти строки
также являются
комментарием
}
...
```

Комментарии могут быть вложенными, но не могут перекрываться:

```
{
начало первого
комментария
(*
второй комментарий
}
неправильный порядок закрытия комментариев
*)
{
начало первого
комментария
(*
второй комментарий
*)
а теперь комментарии закрыты правильно
}
(*
многострочный
комментарий
// внутри него - однострочный
комментарии закрыты правильно
*)
```

Стоит отметить еще один момент. Комбинация символов, начинающаяся с открывающей фигурной скобки и следующим за ним знаком доллара, является *директивой компилятора*. Вы уже встречали такие директивы в предыдущей главе. Ни в коем случае не удаляйте и не изменяйте директивы компилятора, которые вставляет среда Delphi в исходных текстах программ, хотя сразу за ним можно вставить свой комментарий:

```
{$R *.dfm} // Это директива компилятора - не стоит ее трогать.
```

Что такое идентификаторы

Идентификаторы – это слова, которыми программист обозначает любой элемент программы, кроме зарезервированного слова или комментария. Идентификаторы в языке Delphi могут состоять из латинских букв, арабских цифр и знака подчеркивания. Соответственно, недопустимо для идентификатора использование пробелов или включение в идентификатор символов кириллицы (русского алфавита). Идентификаторы могут иметь произвольную длину.

Идентификатор всегда начинается с буквы, за которой могут следовать буквы и цифры. Следует отметить, что буквой считается также и знак подчеркивания. Соответственно, идентификатор может начинаться с этого символа и даже состоять из нескольких символов подчеркивания. А вот пробелы и специальные символы входить в идентификатор не могут. Ниже приведу примеры правильных и неправильных идентификаторов.

Правильные идентификаторы:

```
x
max_value
S1
_my
step100_1
```

Неправильные идентификаторы:

```
begin     // Зарезервированное слово
123_x     // Начинается с цифры
my brush  // Содержит пробел
x$        // Содержит специальный символ $
```

После того как имя идентификатора определено, в тексте программы ему можно присваивать значение. Мы можем использовать идентификаторы для хранения различных данных – строк текста, символов, целых и дробных чисел и т.д. Идентификаторами в общем случае называются константы и переменные.

Константы

Константами называются идентификаторы, определяющие области памяти, которые не могут изменять своего значения. Перед использованием констант они описываются в специальном разделе модуля (под модулем в данной главе мы будем понимать файл с расширением *.pas – файл, в котором обычно пишется текст программы). Раздел описания констант начинается с зарезервированного слова **const**, за которым следует перечень используемых в программе констант и их значения:

```
const
  x_min = 0,000001;
  abc = 'Все права защищены. 2005 г.'
  mb = 1048576;
```

Каждая константа, описываемая в разделе **const**, имеет тип. Тип определяет данные, которые может хранить в себе константа, а также перечень операций с этими данными. В приведенном выше примере константа x_min распознается компилятором как дробное число, abc – как строка символов, mb – как целое число. Тип константы определяется компилятором автоматически, исходя из присвоенного ей в разделе **const** значения. Подробнее о типах константы см. в разделе «Какие бывают типы данных».

При определении констант могут использоваться выражения:

```
const
  a = 5;
  b = 10;
  c = a + b;
```

Кроме того, существуют *типизированные константы*. Такие константы эквивалентны переменным, у которых не изменяется установленное значение. Директива компилятора {$J+} позволяет использовать такие константы для записи:

- {$J+} – типизированные константы доступны только для считения;
- {$J-} – типизированные константы доступны для записи.

Фактически, если включен режим разрешения записи в типизированные константы, мы имеем возможность создать переменную, инициализированную некоторым значением до запуска программы:

```
unit Unit1;
{$J+} // Включаем режим записи типизированных констант.
const
  // Флаг, показывающий, что форма отображается первый раз.
  IsFirst : Boolean = true;
...
// Этот обработчик вызывается при активизации формы.
// Он может вызываться несколько раз, но нам надо
// произвести какие-либо действия только при первом показе формы.
procedure TForm1.FormActivate(Sender: TObject);
begin
  // Если форма активизируется первый раз,
  if IsFirst then
```

```
    begin
      // изменить значение флага.
      IsFirst := false;
      // Выполнить действия при первом показе формы.
      ...
    end;
end;
```

Определенные в разделе **const** константы позволяют использовать имена вместо их значений, что достаточно удобно, – нет необходимости запоминать сложные значения, достаточно использовать короткое имя-идентификатор.

О месте раздела описания констант в структуре модуля см. ниже в пункте «Знакомимся с типовой структурой модуля на языке Delphi».

Переменные

Переменными называются идентификаторы, которые связаны с изменяемыми областями памяти, то есть с таким ее участками, содержимое которых будет меняться в ходе работы программы. Перед использованием переменных они описываются в специальном разделе модуля. Раздел описания переменных начинается с зарезервированного слова **var**, за которым следует перечень переменных и их типов. В отличие от раздела описания констант тип переменной нужно указывать *явно*, так как компилятор не сможет угадать, для хранения каких данных вы будете использовать ту или иную переменную и, соответственно, какой объем памяти необходимо выделить. Ниже приведен пример раздела описания переменных:

```
var
  x,y: integer;
  min_value: real;
  my_str: string;
```

Как видите, имя переменной от типа при ее объявлении отделяется двоеточием (:). Кроме того, если планируется использовать в программе несколько переменных одного типа, то их имена можно указывать через запятую. Подробнее о том, какие бывают типы, см. в разделе «Типы данных».

```
var
  i, j, k : integer;
```

О месте раздела описания переменных в структуре модуля см. ниже в разделе «Знакомимся с типовой структурой модуля на языке Delphi».

Теперь несколько слов скажу о том, как используются переменные. С константами все понятно – их значения определены в разделе описания, и мы можем использовать их имена. Переменным же значения присваиваются непосредственно в разделе реализации (начинающемся со слова `begin`). Чтобы присвоить значение переменной, необходимо указать ее имя, указать оператор присваивания и затем указать присваиваемое переменной значение. В качестве оператора присваивания служит пара символов, состоящая из знака двоеточия и равенства (`:=`). В общем случае оператор присваивания можно записать следующим образом:

```
<переменная>:=<выражение>;
```

О том, какие бывают выражения, можно узнать в разделе «Выражения и операции». В качестве выражения вполне может выступать простое число, строка текста и т.п. Пример присваивания значений переменным приведен ниже:

```
begin
  ...
  x:=0;
  y:=100;
  min_value:=0,5+1/256;
  ...
end;
```

В первых двух случаях выражение представлено простыми числами, а вот в последнем случае уже используется конструкция «классического» выражения – операнды и операции. Результат вычисления выражения и будет записан в переменную `min_value`.

Какие бывают типы данных

Типы данных – специальные конструкции языка, которые рассматриваются компилятором как образцы для создания других элементов программы, таких как переменные, константы и функции. Любой тип определяет две вещи:

- объем памяти, выделяемый для размещения элемента (константы, переменной);
- набор допустимых действий над элементами данного типа.

Первоначально типы как раз и предназначались для того, чтобы программист явно указывал, какого размера память нужна ему и что он с ней собирается делать.

Напомню, что любой определяемый программистом идентификатор должен быть описан в разделе описаний, то есть соответствовать какому-либо типу данных. Далее будут рассмотрены основные типы данных языка Delphi.

Строковый и символьный типы

Знакомство с типами данных мы начнем со строкового типа `String`. Этот тип определяет участок памяти переменной длины, каждый байт которого содержит один символ. Для символов в языке Delphi имеется тип `Char`. Таким образом, тип `String` – это цепочка элементов типа `Char`. Каждый символ типа `String` пронумерован, начиная с единицы. Соответственно, программист может обращаться к любому элементу строки (символу) по его номеру в строке. Следующий фрагмент кода поясняет использование типа `String`:

```
var
  stroka:string;
begin
  ...
  stroka:='Строка текста';
  ...
end;
```

Замечу, что программист обязан объявить любой идентификатор, который он вводит в разделе описаний (**var**). В нашем примере после объявления в разделе **var** идентификатора `stroka` типа `string` будет выделено необходимое количество памяти, и в дальнейшем Delphi будет контролировать использование этого идентификатора. Что значит – контролировать? Это значит, что если мы допустим какую-либо ошибку при использовании этого идентификатора, то нам будет выведено соответствующее сообщение. Например, если далее по тексту программы использовать выражение `1-10*Stroka`, то будет выдано сообщение об ошибке, так как над строками недопустимо использование операции умножения (и других математических операций, за исключением сложения). Соответственно, следующий фрагмент кода демонстрирует использование операции сложения для строковых типов:

```
var
  stroka1,strok2,stroka3:string;
begin
  ...
  stroka1:='Строка ';
  stroka1:='текста';
  stroka3:=stroka1+stroka2;
```

```
  ...
end;
```

В данном случае операция сложения означает объединение строк, то есть добавление строки `stroka2` в конец строки `stroka1`. В результате переменной `stroka3` будет присвоено значение `'строка текста'`.

Кроме того, над строками допустимо использование операций отношения:

- = – равно;
- <> – не равно;
- > – больше;
- < – меньше;
- >= – больше или равно;
- <= – меньше или равно.

Результатом применения операции отношения будет логический тип (описание см. ниже), который имеет два значения: `True` (Истина) и `False` (Ложь).

Еще одним строковым типом данных является строка символов, ограниченная нулем, – `PChar`. На самом деле тип `PChar` является указателем на область памяти, где последний байт содержит символ с номером 0 (напомню, что записывается такой символ с помощью комбинации `#0`).

Для преобразования строки `PChar` в «паскалевскую» строку `String` предназначена специальная функция `StrPas`:

```
function StrPas(const Str: PChar): string;
```

Эта функция на входе принимает указатель на строку, ограниченную нулем, и возвращает строку **`string`**, содержащую те же данные. Так же можно использовать синтаксис преобразования типов:

```
var
  s:string;
  ps:pchar;
begin
  // Конвертация строк с помощью функции
  s:=StrPas(ps);
  // Конвертация строк с помощью преобразования типов
  s:=string(ps);
end;
```

Обратная конвертация выполняется с помощью преобразования типов:

```
ps:=pchar(s);
```

Целые типы

Целые типы используются для хранения и преобразования целых чисел. Язык Delphi предусматривает использование нескольких целых типов, которые отличаются друг от друга диапазоном используемых значений и, соответственно, занимаемым объемом памяти.

Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать один, два, четыре или восемь байтов. Ниже в табл. 3.3 приведены целые типы, используемые в языке Delphi.

Таблица 3.3 ▼ Целые типы, используемые в языке Delphi

Название формата	Диапазон значений	Размер в байтах
`Byte`	0...255	1
`Word`	0...65535	2
`LongWord`	0...4294967295	4
`ShortInt`	-128...127	1
`Integer`	-2147483648...2147483647	4
`Longint`	-2147483648...2147483647	4
`Int64`	$-2^{63}...2^{631}$	8
`Cardinal`	0...4294967295	4

Над целыми типами допустимо применять следующие математические операции:

- `+` – сложение;
- `-` – вычитание;
- `*` – умножение;
- `div` – целочисленное деление;
- `mod` – остаток от деления.

Замечу, что при выполнении операции деления результат может быть дробным, а для этого в языке Delphi используется вещественный тип. Соответственно, поэтому для целого типа приведенные операции деления возвращают только *целочисленный* результат. Использование операций `div` и `mod` можно пояснить на примере. Приведенный ниже фрагмент кода демонстрирует пример использования этих операций.

```
var
  x,y,z:integer;
begin
  ...
  x:=11 div 3;    // x содержит 3
```

```
  y:=11 mod 3;    // y содержит 2
  z:=x div y;     // z содержит 1
  ...
end;
```

Смысл же остальных операций совпадает с общепринятым математическим. Единственное, о чем необходимо помнить, это то, что результат выполнения математической операции не должен выходить за диапазон возможных значений типа (см. табл. 3.3).

Как и к строкам, к целым типам применимы операции отношения.

Вещественный тип

В отличие от рассмотренных выше целых типов, значения которых всегда приравниваются к целым числам, значения *вещественных типов* (Real) всегда определяют число с некоторой точностью, которая зависит от внутреннего формата вещественного числа (табл. 3.4).

Таблица 3.4 ▾ Вещественные типы языка Delphi

Формат	Диапазон	Количество значащих цифр	Длина в байтах
Single	1,5e-45...3,4e38	7-8	4
Real	2,9e-39...1,7e38	11-12	8
Double	5,0e-324...1,7e308	15-16	8
Extended	3,4e-4932...1,1e4932	19-20	10

Как видно из табл. 3.4, вещественное число в Delphi может занимать от 4 до 10 байт. Структура формы представления вещественного числа показана на рис. 3.1.

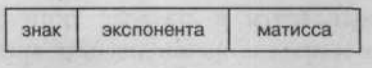

Рис. 3.1 ▾ Представление вещественного числа в памяти компьютера

Поле «знак» представляет собой знаковый разряд числа, «экспонента» – экспоненциальную часть. «Мантисса» может быть представлена 23–63 разрядами двоичных чисел, что обеспечивает точность представления от 6..8 до 19..20 знаков. Десятичная точка подразумевается перед левым разрядом мантиссы. Однако при действиях с числами ее положение сдвигается влево или вправо (в зависимости от порядка числа, хранящегося в экспоненциальной части). Именно поэтому часто действия над вещественными числами называют арифметикой с *плавающей точкой*.

Отметьте для себя, что компьютер *всегда* обрабатывает числа в формате Extended, а все остальные типы в этом случае получаются усечением результатов до нужных размеров и применяются в основном для экономии памяти.

Диапазон

Дапазон (Subbrange) – это специальная конструкция языка Delphi, позволяющая присваивать переменным значения, лежащие в заданном диапазоне. Данный тип задается границами своих значений внутри базового типа, на котором он основан.

Диапазон может задаваться двумя способами – в специальном разделе **type** (см. ниже в разделе «Структура типового модуля программы на языке Delphi») или непосредственно при объявлении переменной. В качестве примера можно привести следующий фрагмент кода:

```
type
  basic_digits = 0..9;
  l_char = 'a'.. 'z';
```

Можно также объявлять диапазон и в разделе **var**:

```
var
  basic_digits : 0..9;
  l_char : 'a'.. 'z';
```

При объявлении диапазона помните, что две точки (..) рассматриваются как один символ, поэтому их нельзя разделять пробелами. Кроме того, левая граница диапазона не должна превышать правую.

Тип «дата-время»

Этот тип данных определяется в разделе описаний стандартным идентификатором TDateTime и предназначен для одновременного хранения даты и времени. Во внутреннем представлении он занимает 8 байт и фактически представляет собой вещественное число, где в целой части хранится дата, а в дробной – время. Дата определяется как количество суток, прошедших с 30 декабря 1899 года, время – как часть суток, прошедших с 0 часов.

Над данным типом определены те же операции, что и с вещественным типом.

Логический тип

Логический тип (Boolean) представляет собой специальную конструкцию, которая может принимать два значения: True (Истина) или False (Ложь).

В разделе описания переменных для объявления данного типа используется слово boolean. Как правило, переменные данного типа используются в выражениях, а также в управляющих конструкциях (операторах) языка Delphi, где проверяется некоторое условие. Пример объявления переменной логического типа приведен ниже.

```
var
  answer_1,answer_2,answer_3 : boolean;
begin
  ...
  answer_1:=true;
  answer_2:=false;
  ...
  answer_3:=answer_1 or answer_2; // answer_3 содержит false
end;
```

Для логического типа допустимо использование логических операций (перечень логических операций см. в разделе «Выражения и операции»).

Перечислимые типы

Перечислимый тип (Enum) представляет собой упорядоченный набор значений, описанный как список идентификаторов:

```
type
  TSound = (tsClick, tsClack, tsClock);
```

Такой тип данных полезен, если вам надо описать переменную, которая принимает ограниченное количество значений.

Набор

Набор (Set) — это коллекция значений некоторого диапазона или перечислимого типа. Тип данных, на основе которого создается набор, не может иметь более 256 значений. Объявление набора выглядит следующим образом:

```
type
  TSound = (tsClick, tsClack, tsClock);
  TSounds = set of TSound;
  TSomeInts = 1..250;
  TIntSet = set of TSomeInts;
```

В программе присваивание значения переменной типа «набор» будет выглядеть следующим образом:

```
var
  s : TintSet;
begin
  s := []; // Пустой набор
  s := [1,2,3]; // Набор, содержащий 3 значения
end;
```

Можно добавлять и удалять значения из переменной типа набор:

```
s := s + [4];
s := s - [1];
```

Проверить, присутствует какой-либо элемент в наборе, можно с помощью конструкции **in**:

```
if 2 in s then ShowMessage('2 присутствует в наборе!');
```

Массивы

Рассмотренные выше простые типы данных позволяют использовать в программе одиночные объекты – различные числа, строки, символы. Язык Delphi также позволяет использовать объекты, которые содержат множество однотипных объектов (чисел, символов, строк). Такие объекты называются *массивами* (Array).

При описании массивов необходимо указать общее количество входящих в массив элементов и тип элементов. Например, следующий фрагмент кода

```
var
  massiv_a:array [0..100] of string;
  massiv_b:array [1..5] of char;
  massiv_c:array [-10..10] of integer;
```

создает три массива соответствующих типов. Как видно из исходного кода, для описания массива используются зарезервированные слова **array** и **of**. За словом **array** следует диапазон чисел, с помощью которого компилятор определит общее количество элементов массива. За словом **of** следует тип содержащихся в массиве элементов.

Чтобы получить доступ к массиву (элементам массива), необходимо в программе указать идентификатор массива и затем в квадратных скобках номер элемента, к которому производится обращение. Следующий фрагмент кода поясняет все вышесказанное.

```
var
  massiv_a:array [0..100] of string;
  massiv_b:array [1..5] of char;
```

```
  massiv_c:array [-10..10] of integer;
begin
  massiv_a[15]:='Пример использования массивов';
  massiv_b[1]:='S';
  massive_c[0]:=24;
end;
```

При использовании массивов помните, что использование элементов массива, индекс которых выходит за пределы объявленного диапазона, недопустимо.

Есть еще один тип массива – *динамический массив* (Dynamic array). Его описание очень похоже на описание обычного, статического массива, однако в нем отсутствует указание границ массива:

```
var
  dyn_massiv:array of integer;
```

Правила работы с ним точно такие же, как и с обычным массивом, то есть обращение к его элементам производится по индексу, а выход за границу массива приводит к возникновению ошибки. Но как же тогда указать в программе нужную длину массива и узнать ее, если границы массива не указываются при его описании, спросите вы? Для этого существуют специальные процедуры и функции, перечисленные в табл. 3.5.

Таблица 3.5 ▾ Функции работы с динамическими массивами

Имя	Описание
`function Length(S): Integer;`	Возвращает количество элементов в массиве или символов в строке
`procedure SetLength (var S; NewLength: Integer);`	Устанавливает новую длину массива или строки, при этом сохраняет те элементы, которые были в массиве или строке до ее изменения
`function Low(X);`	Возвращает индекс нижней границы массива
`function High(X);`	Возвращает индекс верхней границы массива

Надо только помнить, что при описании динамический массив всегда имеет нулевую длину. Кстати, хотя функция Low для любого массива вернет индекс первого элемента, но все динамические массивы имеют индекс первого элемента, равный нулю. А теперь рассмотрим пример использования динамического массива и функций работы с ним:

```
procedure TForm1.FormActivate(Sender: TObject);
var
```

```
  i : integer;
  a : array of integer;
  sum : integer;
  avg : double;
  msg : string;
begin
  // Выделяем 10 элементов.
  setlength(a,10);
  // Заполняем массив случайными числами от 1 до 100.
  for i := Low(a) to High(a) do
    a[i] := Random(100)+1;
  // Считаем сумму элементов массива.
  sum := 0;
  for i := Low(a) to High(a) do
    sum := sum + a[i];
  // Считаем среднее значение среди элементов массива.
  avg := sum / Length(a);
  // Формируем и выводим сообщение с результатами.
  msg := 'Массив: ';
  for i := Low(a) to High(a)-1 do
    msg := msg + IntToStr(a[i]) + ',';
  msg := msg + IntToStr(a[High(a)]) + #13#10;
  msg := msg + 'Сумма элементов = ' + IntToStr(sum) + #13#10;
  msg := msg + 'среднее значение = ' + FloatToStr(avg);
  ShowMessage(msg);
end;
```

Результат работы этой небольшой программы вы можете видеть на рис. 3.2.

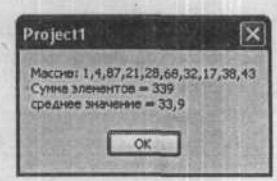

Рис. 3.2 ▼ Пример работы с динамическим массивом

Записи

Запись (Record) – сложная структура данных, состоящая из фиксированного количества объектов, называемых *полями* записи. В отличие от массивов поля записи могут быть разного типа. Чтобы можно было ссылаться на тот

или иной элемент, поля записи именуются. Структура объявления записи такова:

```
<имя_типа_записи>=record <список_полей> end;
```

Имя типа записи – идентификатор, который будет использоваться для доступа к данному типу. **Record** и **end** – зарезервированные слова, означающие начало описания записи и его конец. Список полей представляет собой последовательность описания идентификаторов, очень похожий на описание обычных типов в разделе **var**. Пример описания типа приведен ниже:

```
type
  Personal = record
    Name:string;
    Age:Byte;
    Payment:integer;
  end;
```

В этом примере создается запись `Personal` – запись, содержащая информацию о сотруднике. Такой информацией в данном случае является его имя, возраст и оклад. Соответственно, определяются три поля записи – поле `Name` типа `string` для хранения имени, поле `Age` типа `Byte` – для возраста и поле `Payment` типа `Integer` для оклада. После того как создано описание подобного типа, необходимо создать объект такого типа – только тогда подобную структуру можно использовать. Для этого в разделе **var** достаточно создать идентификатор созданного нами типа `Personal`:

```
var
  boss:Personal;                          // Возможно использование
                                          // одиночного объекта типа
                                          // запись
employment:array [1..100] of Personal; // или использование множества
                                          // объектов.
```

Доступ в тексте программы к объявленным идентификаторам можно получить следующим образом:

```
begin
  Boss.Name:='Иван Петров';
  Boss.Age:=30;
  Boss.Payment:=40000;
  employment[10].Name:='Федор Сумкин';
```

```
  employment[10].Age:= Boss.Age;                    // Возраст Сумкина 30.
  employment[10].Payment:= Boss.Payment div 4; // Оклад Сумкина равен
                                                    // 10000.
end;
```

Изучаем основные типы выражений и операции

Основными элементами, из которых состоит исполняемая часть программы, являются константы и переменные, а также различные действия, которые с ними производятся. Каждый из этих элементов характеризуется своим значением и принадлежит к какому-либо типу данных. С помощью знаков операций и скобок из них можно составлять *выражения*, которые фактически представляют собой правила получения новых значений. В общем случае выражения состоят из нескольких элементов (операндов) и знаков операций. Тип получаемого результата определяется типом операндов и видом примененных к ним операций.

В языке Delphi определены следующие операции:

- унарные (например, `not`)
- мультипликативные (*, /, `div`, `mod`, `and`);
- аддитивные (+, –, `or`);
- отношения (=, <>, >, <, >=, <=).

Приоритет операций убывает в указанном порядке, то есть высшим приоритетом обладают унарные операции, низшим – операции отношения. Порядок выполнения нескольких операций равного приоритета устанавливается компилятором из условия оптимизации кода и не обязательно слева направо. А при вычислении логических выражений операции равного приоритета всегда вычисляются слева направо. Ниже в табл. 3.6 приведены некоторые основные операции, которые можно использовать в языке Delphi.

Таблица 3.6 ▼ Основные операции, используемые в языке Delphi

Операция	Какое действие производится	Тип используемых операндов	Тип получаемого результата
+	сложение	любой целый	наименьший целый
+	сложение	любой вещественный	extended
+	сцепление строк	строковый	строковый

Таблица 3.6 ▾ Основные операции, используемые в языке Delphi (окончание)

Операция	Какое действие производится	Тип используемых операндов	Тип получаемого результата
-	вычитание	любой целый	наименьший целый
-	вычитание	любой вещественный	extended
*	умножение	любой целый	наименьший целый
*	умножение	любой вещественный	extended
/	деление	любой вещественный	extended
div	целочисленное деление	любой целый	наименьший целый
mod	остаток от деления	любой целый	наименьший целый
not	отрицание	логический	логический
not	отрицание	любой целый	наименьший целый
and	логическое И	логический	логический
and	логическое И	любой целый	наименьший целый
or	логическое ИЛИ	логический	логический
or	логическое ИЛИ	любой целый	наименьший целый
shl	левый сдвиг	любой целый	наименьший целый
shr	правый сдвиг	любой целый	наименьший целый
=	равно	любой простой или строковый	логический
<>	не равно	любой простой или строковый	логический
<	меньше	логический	логический
>	больше	логический	логический
<=	меньше или равно	логический	логический
>=	больше или равно	логический	логический

Немного подробнее остановимся на логических операциях языка Delphi. В языке существуют следующие логические операции:

- not – логическое НЕ (отрицание);
- and – логическое И (логическое умножение);
- or – логическое ИЛИ (логическое сложение)
- xor – исключающее ИЛИ.

Логические операции можно применять к операндам логического или целого типов. Если операнды – целые числа, то результат логической операции – тоже целое число, биты которого формируются из битов операндов по приведенным в табл. 3.7 правилам.

Таблица 3.7 ▾ Логические операции над данными целого типа

Операнд 1	Операнд 2	not	and	or	xor
1	отсутствует	0	отсутствует	отсутствует	отсутствует
0	отсутствует	1	отсутствует	отсутствует	отсутствует

Таблица 3.7 ▼ Логические операции над данными целого типа (окончание)

Операнд 1	Операнд 2	not	and	or	xor
0	0	отсутствует	0	0	0
0	1	отсутствует	0	1	1
1	0	отсутствует	0	1	1
1	1	отсутствует	1	1	0

Логические операции над данными логического типа дают результат, сформированный по правилам, указанным в табл. 3.8

Таблица 3.8 ▼ Логические операции над данными логического типа

Операнд 1	Операнд 2	not	and	or	xor
true	отсутствует	false	отсутствует	отсутствует	отсутствует
false	отсутствует	true	отсутствует	отсутствует	отсутствует
false	false	отсутствует	false	false	false
false	true	отсутствует	false	true	true
true	false	отсутствует	false	true	true
true	true	отсутствует	true	true	false

Знакомимся с операторами языка Delphi

С некоторыми *операторами* языка Delphi мы уже познакомились по ходу прочтения книги и разбору примеров. Это, например, оператор присваивания (:=), с помощью которого мы присваивали значения переменным. Для составления исходного кода программ нам потребуется знание некоторого количества основных операторов языка (инструкций). Объем данной книги не позволяет подробно рассмотреть все инструкции языка, поэтому будут рассмотрены только самые основные операторы.

Составной оператор begin

Составной оператор – последовательность произвольных операторов, заключенных в операторные скобки, – зарезервированные слова `begin` и `end`. Язык программирования Delphi не предполагает никаких ограничений по использованию этого оператора – составной оператор может содержать сколько угодно вложенных составных операторов.

Как правило, подобный оператор используется в описанных ниже операторах, где требуется выполнять определенную последовательность действий.

Условный оператор

Условный оператор позволяет проверить некоторое условие и в зависимости от результатов выполнить то или иное действие. Таким образом, условный оператор – это средство ветвления вычислительного процесса.

Структура условного оператора имеет следующий вид:

```
if <условие> then <оператор1> else <оператор2>;
```

Здесь **if**, **then**, **else** – зарезервированные ключевые слова (в переводе с англ. означающие: если, то, иначе), <условие> – произвольное выражение логического типа, <оператор1> и <оператор2> – любые операторы языка Delphi (или последовательности операторов, заключенные в конструкции begin-end). Допускается также использовать и сокращенный вариант условного оператора:

```
if <условие> then <оператор>;
```

Работает условный оператор по следующему алгоритму, представленному на рис. 3.3. Сначала вычисляется условное выражение (<условие>). Если результатом логического выражения будет значение True (Истина), то выполняется оператор, стоящий после слова **then**. Если же результатом логического выражения будет значение False (Ложь), то будет выполнен оператор, стоящий после слова **else**. Сокращенный вариант условного оператора работает аналогично (рис. 3.4), отличие состоит в том, что оператор после слова then не выполняется (пропускается), если значение условия равно False.

Рассмотрим вариант использования условного оператора:

```
var
  x,y,value:integer;
begin
...
if x>y then value:=x+y else value:=x-y;
...
end;
```

В приведенном примере значение переменной value определяется в зависимости от результата сравнения переменных x и y. Если x>y, то в переменную value запишется сумма этих двух чисел, в противном случае – разность.

Условие в условном операторе может быть составным. В этом случае отдельные условия «закрываются» в круглые скобки и объединяются логическими

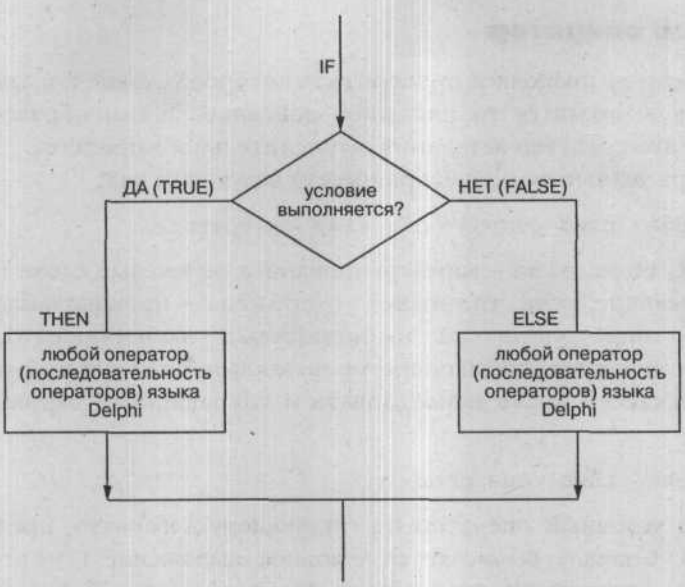

Рис. 3.3 ▼ Блок-схема работы условного оператора `if-then-else`

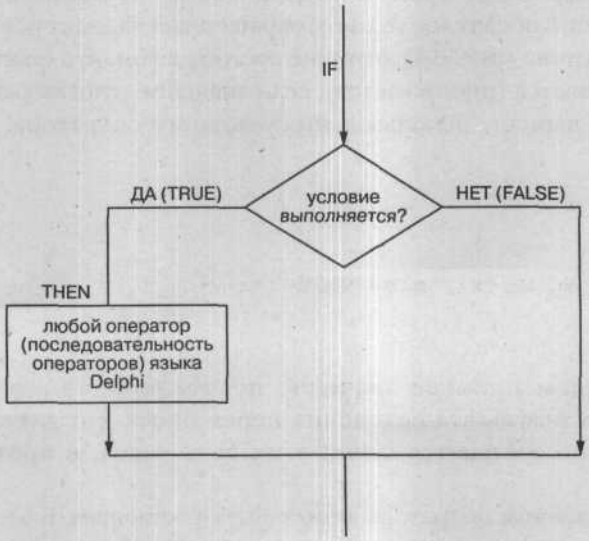

Рис. 3.4 ▼ Блок-схема работы сокращенного оператора `if-then`

операторами not, or, and или xor. Формат такого условия выглядит следующим образом:

```
 (<условие1>) <логический_оператор1> (<условие2>)
<логический_оператор2> … <логический_операторN-1> (<условиеN>)
```

Пример использования сложного условия приведен ниже:

```
var
  age:integer;
begin
...
// Если возраст от 18 до 50,
if (age>=18) and (age<=50) then … //выполнить действия.
...
end;
```

Операторы повторений

В языке Delphi имеются специальные операторы, позволяющие выполнять идентичные операции некоторое количество раз. Такие операторы называются операторами повторения, или *операторами цикла*. Мы рассмотрим несколько подобных операторов.

Счетный оператор цикла **for-to-do** имеет следующую структуру:

```
for <параметр_цикла>:=<начальное_значение> to <конечное_значение>
do <оператор>;
```

Здесь **for**, **to**, **do** – зарезервированные слова (в переводе с англ. – для, до, выполнить), <параметр_цикла> – переменная типа Integer, <начальное_значение> – начальное значение или выражение того же типа, <конечное_значение> – конечное значение или выражение того же типа, <оператор> – произвольный оператор Delphi (или последовательность операторов, заключенная в конструкцию **begin-end**).

Оператор **for-to-do** работает следующим образом:

1. Сначала вычисляется выражение <начальное_значение>.
2. Параметру цикла присваивается начальное значение.
3. Осуществляется проверка <параметр_цикла> <= <конечное_значение>. Если условие не выполняется, то оператор цикла завершает свою работу.
4. Выполняется <оператор> после слова do.
5. Параметр цикла наращивается на единицу. Далее снова выполняются пункты 3–5 до тех пор, пока параметр цикла не достигнет конечного значения.

Рис. 3.5 ▼ *Блок-схема работы оператора цикла* `for-to-do`

Схематично работа оператора цикла `for-to-do` представлена на рис. 3.5.

Приведенный ниже фрагмент исходного кода демонстрирует вариант применения данного оператора. В результате выполнения этого оператора цикла в переменную s запишется сумма чисел от 1 до 5.

```
var
  i,s:integer;
```

```
begin
  ...
  s:=0;
  for i:=1 to 5 do s:=s+i;
  ...
end;
```

Также при использовании оператора **for-to-do** необходимо учесть следующее. Во-первых, условие, управляющее работой этого оператора, проверяется перед выполнением оператора. Соответственно, если это условие заведомо не выполняется, то и оператор цикла не будет выполнен ни разу. Во-вторых, вместо зарезервированного слова **to** допускается использование слова downto. Отличие состоит в том, что на пятом шаге выполнения оператора (см. выше) параметр цикла *уменьшает* свое значение на единицу, а условие, управляющее работой цикла, меняет знак отношения с >= на <=.

Рассмотрим следующий оператор цикла – оператор цикла *с предпроверкой условия*. Структура данного оператора следующая:

```
while <условие> do <оператор>;
```

Здесь **while**, **do** – зарезервированные слова (в переводе с англ. – пока [выполняется условие], делать), <условие> – выражение логического типа, <оператор> – любой оператор языка Delphi (последовательность операторов, заключенная в конструкцию **begin-end**).

Оператор **while-do** работает следующим образом:

1. Вычисляется значение выражения <условие>.
2. Если выражение <условие> выполняется (имеет значение True), то выполняются инструкции цикла.
3. Если выражение условия не выполняется (имеет значение False), то инструкции цикла не выполняются, и выполнение цикла заканчивается.

Схема работы оператора **while-do** показана на рис. 3.6.

Приведенный ниже код выполняет те же действия, что и в предыдущем примере, однако здесь используется конструкция **while-do**:

```
var
  i,s:integer;
begin
  ...
  i:=1
  s:=0;
  while i<=5 do
```

Рис. 3.6 ▼ Блок-схема работы оператора `while-do`

```
    begin
      s:=s+i;
      i:=i+1;
    end;
  ...
end;
```

Здесь, как и в предыдущем случае, если условие неверно, то и оператор цикла не будет выполняться.

Рассмотрим еще один оператор цикла – оператор **repeat-until** – *цикл с постпроверкой условия*. Структура оператора следующая:

```
repeat <тело_цикла> until <условие>;
```

Здесь **repeat** и **until** – зарезервированные слова (в переводе с англ. – повторять [до тех пор], пока [не будет выполнено условие]), <тело_цикла> – произвольная последовательность операторов Delphi, <условие> – выражение логического типа.

Основное отличие этого оператора цикла от предыдущих состоит в том, что <тело_цикла> всегда выполнится хотя бы один раз.

Работает оператор **repeat-until** следующим образом:

1. Сначала выполняется оператор (последовательность операторов) между словами **repeat** и **until**.
2. Вычисляется значение выражения <условие>. Если условие не выполняется (имеет значение False), то повторно выполняются инструкции цикла.
3. Если условие выполняется (имеет значение True), то выполнение цикла прекращается.

Схема работы оператора **repeat-until** приведена на рис. 3.7.

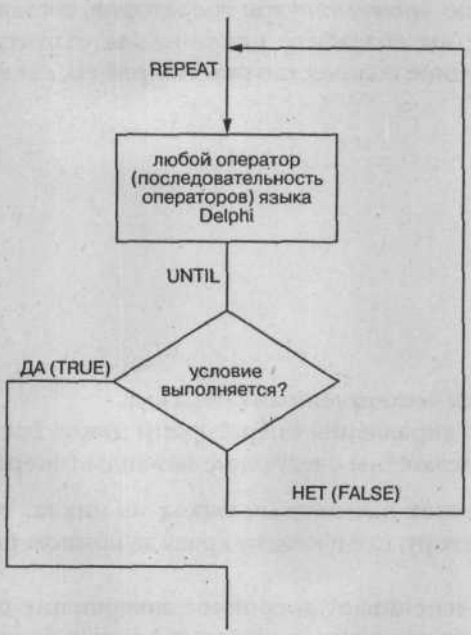

Рис. 3.7 ▾ Блок-схема работы оператора **repeat-until**

Ниже приведен уже знакомый нам пример с применением конструкции **repeat-until**:

```
var
  i,s:integer;
begin
  ...
  i:=1
  s:=0;
```

```
  repeat
    begin
      s:=s+i;
      i:=i+1;
    end
  until i>5
  ...
end;
```

Замечу, что для правильного выхода из циклов **while-do** и **repeat-until** условие выхода должно *меняться* внутри операторов, составляющих тело цикла. В противном случае вы создадите циклические структуры, которые будут выполняться бесконечное количество раз. Например, следующие конструкции циклов

```
while True do
begin
...
end;
  и
repeat
...
until False;
```

никогда не завершатся «естественным» образом.

Для более гибкого управления операторами цикла `for`, `while` и `repeat` в состав языка Delphi включены следующие команды (операторы):

- `Break` – реализует *немедленный* выход из цикла, то есть передачу управления оператору, следующему сразу за концом циклического оператора;
- `Continue` – обеспечивает досрочное завершение очередного прохода цикла, эквивалент передачи управления в самый конец циклического оператора;

Оператор выбора

Оператор выбора позволяет выбрать одно из нескольких возможных продолжений программы. Параметром, по которому осуществляется выбор, служит *ключ выбора* – выражение целого, символьного или логического типа. Структура оператора выбора следующая:

```
case <ключ_выбора> of <список выбора> [else < оператор[ы] >]
end;
```

Здесь **case**, **of**, **else**, **end** – зарезервированные слова (в переводе с англ. – выбрать, из, иначе, конец [оператора выбора]), <ключ_выбора> – ключ выбора, выражение порядкового типа, <список_выбора> – одна или более конструкций следующего вида:

```
<константа_выбора>:< оператор>;
```

Здесь <константа_выбора> – константа того же типа, что и выражение <ключ_выбора>, <оператор> – любой оператор языка Delphi (последовательность операторов).

Оператор выбора работает следующим образом:

1. Вычисляется выражение <ключ_выбора>.
2. Среди последовательности операторов <список_выбора> отыскивается такой, которому предшествует константа, равная вычисленному значению.
3. Если в списке выбора такой оператор найден, то он выполняется, после чего оператор выбора завершает свою работу.
4. Если в списке выбора не будет найдена константа, равная вычисленному выражению, то будут выполняться операторы, стоящие за ключевым словом **else**.

Схематично работа оператора выбора представлена на рис. 3.8.

Замечу, что оператор **else** использовать не обязательно. В этом случае при отсутствии в списке выбора нужной константы оператор выбора просто завершит свою работу (рис. 3.9).

Ниже приведено несколько вариантов использования оператора **case**. В качестве примера мы определим, является ли день, порядковый номер которого хранится в переменной num, рабочим или выходным. Соответственно, результат мы будем заносить в качестве строки в переменную msg. Итак, самый простой вариант применения конструкции **case** приведен ниже:

```
case num_day of
  1: msg:='рабочий день';
  2: msg:='рабочий день';
  3: msg:='рабочий день';
  4: msg:='рабочий день';
  5: msg:='рабочий день';
  6: msg:='выходной день - суббота';
  7: msg:='выходной день - воскресенье';
end;
```

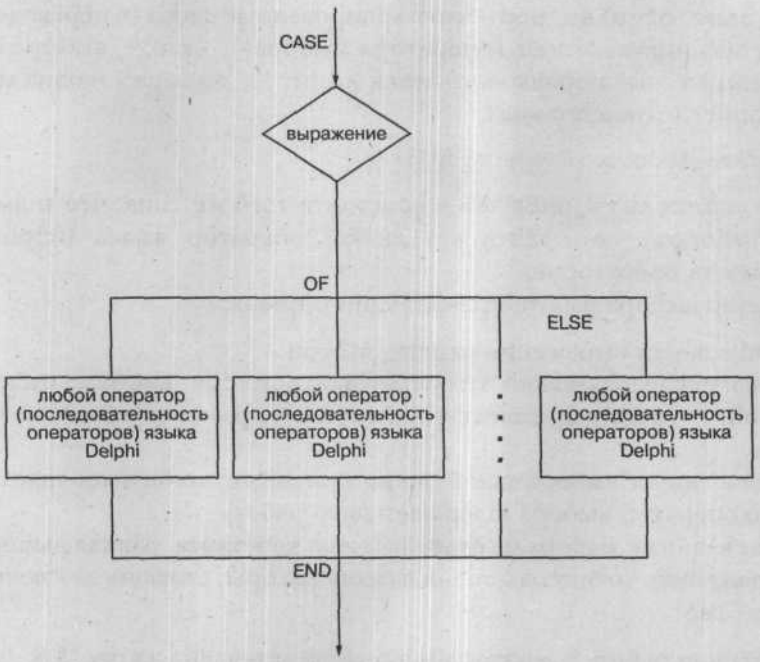

Рис. 3.8 ▼ Блок-схема работы оператора выбора

Очевидно, что приведенный пример даже на первый взгляд не очень хорош. А вот следующая конструкция уже выглядит гораздо короче, хотя и выполняет абсолютно те же действия:

```
case num_day of
  1..5: msg:='рабочий день';
  6: msg:='выходной день - суббота';
  7: msg:='выходной день - воскресенье';
end;
```

Как вариант конструкции с использованием ключевого слова **else** можно привести и такой вариант записи этого примера:

```
case num_day of
  6: msg:='выходной день - суббота';
  7: msg:='выходной день - воскресенье';
  else msg:='рабочий день';
end;
```

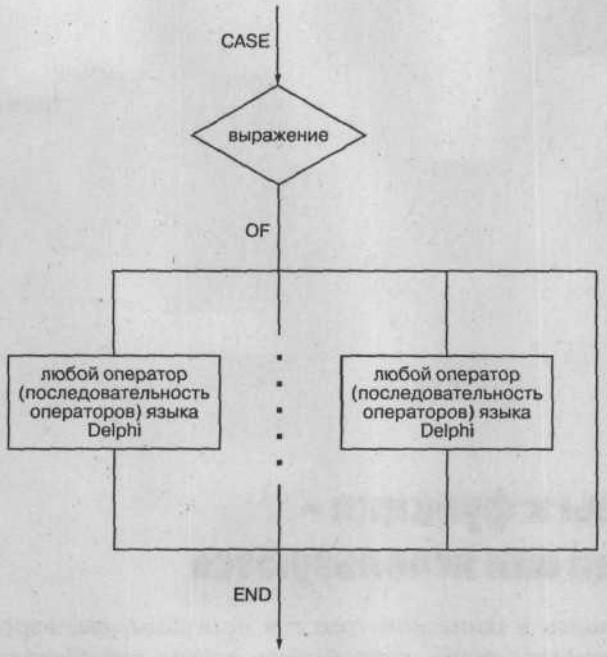

Рис. 3.9 ▾ Блок-схема работы сокращенного варианта оператора выбора

Метки и операторы перехода

Оператор перехода позволяет передать управление в место программы, обозначенное меткой. Оператор перехода имеет следующий вид:

```
goto <метка>;
```

Здесь `goto` – зарезервированное слово (в переводе с англ. – перейти [на метку]), `<метка>` – идентификатор, описанный в специальном разделе `label`.

Метка (Label) – это произвольный идентификатор программы, позволяющий именовать некоторый оператор программы и таким образом ссылаться на него. Метка располагается перед помеченным оператором и отделяется от него знаком двоеточия. Допускается помечать операторы и несколькими метками – в этом случае они отделяются друг от друга двоеточием. Описываются метки в специальном разделе **`label`**, предшествующему разделу описания переменных **`var`**.

```
label
  m1,m2,m3;
begin
  ...
  goto m1;
...
m1: ...
  ...
m2:m3: ...
  ...
  goto m2;
...
  goto m3;
  ...
end;
```

Процедуры и функции – где и когда они используются

Попрактиковавшись в написании текстов программ, вы, вероятно, ощутили некоторую сложность: очень часто бывает трудно разобраться в тексте программы. Для облегчения написания больших и сложных программ, а также для лучшей их структурированности в языке Delphi существуют важные инструменты – процедуры и функции. Процедуры и функции представляют собой во многом *самостоятельные* фрагменты программы, которым передается управление. По завершении работы процедур и функций управление возвращается основной программе. Такой подход позволяет в процедурах и функциях (довольно часто их называют *подпрограммами*) реализовать некоторые детали того или иного алгоритмического действия, поэтому изменение этих деталей, например, в процессе отладки, как правило, не приводит к изменениям основной программы.

Попробуем поближе познакомимся с понятием процедур и функций.

Процедурой (Procedure) в Delphi называется особым образом оформленный фрагмент программы, имеющий собственное имя. Упоминание этого имени в программе приводит к тому, что процедура *активизируется*, то есть ей передается управление. Сразу после передачи управления процедуре начинают выполняться входящие в нее операторы. После выполнения последнего из них управление передается обратно в основную программу, и выполняются операторы, стоящие непосредственно за оператором вызова процедуры. Все вышесказанное иллюстрирует рис. 3.10.

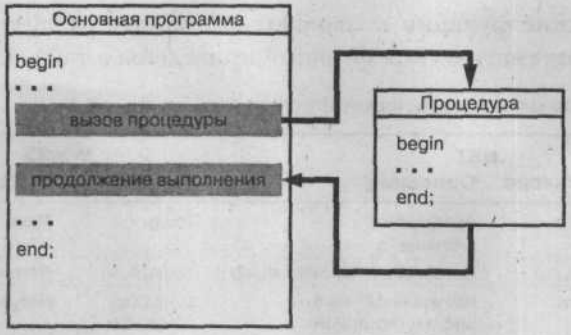

Рис. 3.10 ▾ Взаимодействие основной программы и процедуры

Для обмена информацией между основной программой и процедурой используются один или несколько *параметров вызова*. Допускается использовать процедуры и без параметров вызова.

Теперь выясним отличие процедуры от функции. *Функция* (Function) отличается от процедуры тем, что результат ее работы возвращается в виде значения этой функции. Процедура же никакого значения по завершении своей работы не возвращает. Таким образом, вызов функции может использоваться наряду с другими операндами при вычислении выражений.

В язык Delphi встроен достаточно внушительный набор функций и процедур для работы с самыми различными типами данных. Такие процедуры и функции называются стандартными. Описание некоторых стандартных процедур и функций для работы с описанными выше типами данных приведены ниже.

Некоторые стандартные процедуры и функции

С примерами стандартных функций мы уже сталкивались. Вспомните, например, преобразование числа в строку – для этого мы использовали функцию `IntToStr` при программировании для Win32 и функцию `ToString` при программировании для .NET. Эти функции входят в состав библиотеки среды разработки Borland Delphi 2005. Наличие таких стандартных функций существенно облегчает разработку программ, ведь в противном случае нам пришлось бы писать подобные функции самостоятельно. Перечень доступных функций зависит от того, какие модули и библиотеки вы используете. Поэтому, в зависимости от перечня модулей, которые указаны в секции `uses`, будут доступны те или иные процедуры и функции.

Начнем рассмотрение с математических функций.

Математические функции позволяют выполнять различные вычисления. Некоторые из математических функций приведены в табл. 3.9.

Таблица 3.9 ▼ Основные математические функции

Функция	.NET		Win32	
	Синтаксис	Описание	Синтаксис	Описание
Абсолютное значение	`abs(n)`	Абсолютное значение `n`	То же	То же
Логарифм	`log(n,m)`	Логарифм `n` по основанию `n`	`logN(n,m)`	Логарифм `m` по основанию `n`
Натуральный логарифм	`log(n)`	Натуральный логарифм `n` (логарифм *по основанию e*, где e = 2,718281828459)	`lnXp1(n)` логарифм `n`	Натуральный
Десятичный логарифм	`log10(n)`	Десятичный логарифм `n`	`log10(n)`	Десятичный логарифм `n`
Синус	`sin(a)`	Синус угла `a`, заданного в радианах	`sin(a)`	Синус угла `a`, заданного в радианах
Косинус	`cos(a)`	Косинус угла `a`, заданного в радианах	`cos(a)`	Косинус угла `a`, заданного в радианах
Тангенс	`tan(a)`	Тангенс угла `a`, заданного в радианах	`tan(a)`	Тангенс угла `a`, заданного в радианах
Арксинус	`asin(a)`	Арксинус `a` (угол в радианах), синус которого равен `a`	`arcsin(a)`	Арксинус `a` (угол в радианах), синус которого равен `a`
Арккосинус	`acos(a)`	Арккосинус `a` (угол в радианах), синус которого равен `a`	`arcos(a)`	Арккосинус `a` (угол в радианах), синус которого равен `a`
Арктангенс	`atan(a)`	Арктангенс `a` (угол в радианах), синус которого равен `a`	`arctan(a)`	Арктангенс `a` (угол в радианах), синус которого равен `a`
Квадратный корень	`sqrt(n)`	Квадратный корень из `n`	То же	То же
Случайное число	`next (n_random)`	Случайное число в диапазоне `0…(n_random-1)`	`random(n)`	Случайное число из диапазона `0..n`

Ниже в табл. 3.10 приведены некоторые функции преобразования типов.

Таблица 3.10 ▼ Основные функции преобразования типов

Функция	.NET		Win32	
	Синтаксис	Результат	Синтаксис	Результат
Преобразование строки в целое число	`ToByte(s)` `ToInt16(s)` `ToInt32(s)` `ToInt64(s)`	Целое значение типа `Byte`, `Int16`, `Int32`, `Int64` – результат преобразования строки `s` в число	`StrToInt(s)`	Целое значение типа `integer` – результат преобразования строки `s` в число

Таблица 3.10 ▾ Основные функции преобразования типов (окончание)

Функция	.NET		Win32	
	Синтаксис	Результат	Синтаксис	Результат
Преобразование строки в вещественное число	`ToSingle(s)` `ToDouble(s)`	Дробное значение типа `Single` или `Double` – результат преобразования строки `s` в число	`FloatToInt(s)`	Дробное значение типа `real` – результат преобразования строки `s` в число
Преобразование числа в строку	`ToString(s)`	Строка, параметр использовать необязательно. При указании параметра задается формат отображения	`FloatToStr(n)` `IntToStr(n)`	Строка, результат преобразования числа `n`

В табл. 3.11 приведены некоторые функции для работы со строками.

Таблица 3.11 ▾ Основные функции для работы со строками

Функция	.NET		Win32	
	Синтаксис	Результат	Синтаксис	Результат
Длина (количество символов) строки	`s.length`	Возвращает длину строки `s`, результат – целое число	`Length(s)`	Возвращает длину строки `s`, результат – целое число
Выделение подстроки из строки	`s.substring(i,n)`	Выделяет из строки `s` `n` символов, начиная с `i`-го	`Copy(str,i,n)`	Выделяет из строки `str` `n` символов, начиная с `i`-го
Вставка символов в строку	`s.insert(i,str)`	Вставляет строку `str` в строку `s`, начиная с `i`-го символа	`insert(str,s,i)`	Вставляет строку `str` в строку `s`, начиная с `i`-го
Удаление символов из строки	`s.remove(i,n)`	Удаляет из строки `s` `n` символов, начиная с `i`-го	`delete(str,i,n)`	Удаляет из строки `str` `n` символов, начиная с `i`-го
Замена строчных символов на прописные	`s.ToUpper`	Заменяет в строке `s` все строчные символы прописными	`AnsiUpperCase(s)`	Заменяет в строке `s` все строчные символы прописными
Замена прописных символов строчными	`s.ToLower`	Заменяет в строке `s` все прописные символы строчными	`AnsiLowerCase(s)`	Заменяет в строке `s` все прописные символы строчными

Далее в табл. 3.12 приведены основные функции для работы с датой и временем.

Таблица 3.12 ▼ Функции для работы с датой и временем

Функция	.NET		Win32	
	Синтаксис	Результат	Синтаксис	Результат
Текущая дата	`datetime.now`	Возвращает дату (структуру `datetime`)	`dateof(value)`	Возвращает дату (структуру `datetime`)
Текущий день месяца	`d.day`	Возвращает текущий день (1..31) месяца даты `d`	`dayof(value)`	Возвращает текущий день месяца 1..31 даты `value`
Номер дня даты	`d.dayofyear`	Возвращает порядковый номер дня (1..366) в году даты `d`	`dayoftheyear (value)`	Возвращает порядковый номер дня (1..366) в году даты `value`
Месяц	`d.month`	Возвращает месяц указанной даты `d`	`monthof(value)`	Возвращает месяц указанной даты `value`
Год	`d.year`	Возвращает год указанной даты `d`	`yearof(value)`	Возвращает год указанной даты `value`
Часы	`d.hour`	Возвращает час указанной даты `d`	`hourof(value)`	Возвращает час указанной даты `value`
Минуты	`d.minute`	Возвращает минуты указанной даты `d`	`minuteof(value)`	Возвращает минуты указанной даты `value`

Более подробную информацию по использованию стандартных функций можно получить в справочной системе. Как правило, достаточно ввести любую из приведенных функций в строке поиска и вам будет выведен соответствующий этой группе функций раздел, где подробно описана каждая функция.

Как правило, в большинстве случаев стандартных функций не хватает – тогда программисту приходится писать функции и процедуры самостоятельно. О том, как это делается, вы узнаете в следующем разделе.

Процедуры и функции, определяемые программистом

Многие примеры программ, приведенные в книге, достаточно просты, поэтому в таких программах можно обойтись без процедур и функций, определяемых программистом. Однако вспомните, когда мы создавали первые проекты, то уже тогда столкнулись с понятием процедуры обработки события. Программа делилась на фрагменты – процедуры обработки того или иного события.

Описание процедуры начинается зарезервированным словом **procedure**. Сразу за этим словом следует *имя* процедуры и список формальных параметров. Список параметров заключается в скобки и для каждого параметра указывается его тип.

Зарезервированное слово **procedure**, имя процедуры и список ее параметров называются *заголовком* процедуры, или ее *сигнатурой*. За заголовком процедуры

следует тело процедуры – описание типов, меток, констант, переменных и, собственно, список операторов, заключенный в конструкцию **begin** и **end**:

```
procedure <имя_процедуры>(<список_параметров>);
type <описание_типов>;
label <описание_меток>;
const <описание_констант>;
var <описание_переменных>;
begin
  ...
  // Здесь располагаются операторы процедуры.
  ...
end;
```

Пример описания функции приведен ниже. Заголовок функции начинается с ключевого слова **function**, за которым следует имя функции и список формальных параметров, заключенный в скобки. За списком формальных параметров следует тип возвращаемого значения, отделяемый от скобок двоеточием:

```
function <имя_функции>(<список_параметров>):<тип_возвращаемого_
значения>;
type <описание_типов>;
label <описание_меток>;
const <описание_констант>;
var <описание_переменных>;
begin
  ...
  // Здесь располагаются операторы функции.
  ...
end;
```

Как уже было сказано, функция от процедуры отличается тем, что у нее есть возвращаемое значение. На самом деле, Delphi кроме всего прочего создает и описывает в функции переменную `Result`, по типу данных совпадающую с типом данных, возвращаемых функцией. Таким образом, вы можете как указывать возвращаемое значение старым образом, как в Паскале, присваивая результат имени функции, а можете для этого использовать переменную `Result`.

Теперь скажем насколько слов о списке параметров процедур и функций. В общем виде он выглядит следующим образом:

```
[var | out | const] <имя_параметра> : <тип_параметра>, ...
```

то есть состоит из списка конструкций описания параметров, перечисленных через запятую. Если посмотрите на этот формат внимательно, то увидите, что он очень похож на описание переменных. Действительно, для процедуры или

функции параметры являются *локальными* переменными, то есть видны только внутри нее самой. Нам осталось разобраться только с модификаторами, которые могут стоять перед параметром (или отсутствовать вообще). Табл. 3.13 сделает смысл модификаторов понятным и для вас.

Таблица 3.13 ▼ Модификаторы параметров процедур и функций

Модификатор	Описание
<отсутствует>	Значение параметра копируется в локальную переменную. Вы можете изменять записанное в параметре значение и использовать его в вычислениях, но при выходе из процедуры или функции значение параметра останется таким же, как и при ее вызове
var	В процедуру или функцию копируется указатель на переданный параметр. При вызове процедуры или функции на месте var-параметра должна быть переменная. При изменении значения параметра в теле процедуры или функции при возврате из нее в вызвавшую программу вернется новое значение
out	Параметр для возвращаемого значения. При вызове процедуры или функции содержимое этого параметра будет очищено. В остальном работа out-параметра аналогична var-параметру
const	Данный тип параметра аналогичен локальной константе. Этот параметр работает аналогично случаю, когда не указан вообще никакой модификатор, за исключением того, что вы не можете присваивать ему значение в теле процедуры или функции

Знакомимся с типовой структурой модулей на языке Delphi

Любой модуль (файл с расширением *.pas) на языке Delphi построен по типу процедуры (иногда его еще называют *главной процедурой*). Ниже приведена структура типового модуля Delphi. Синтаксис отдельных элементов может отличаться в зависимости от реализации (например, .NET или Win32), но в той или иной степени эти разделы присутствуют всегда.

Итак, типовая структура модуля (процедуры) на языке Delphi выглядит следующим образом:

```
unit <Имя_программы>;
// Далее следует описательная часть модуля - начинается со слова
// interface.
Interface
// Раздел описания используемых модулей - начинается со слова uses.
uses <список_используемых_модулей_через_запятую>;
// Раздел описания используемых типов данных - начинается со слова
// type.
```

```
type <список_используемых_типов>;
// Раздел описания констант.
const <список_констант>;
// Раздел_описания_переменных - начинается со слова var.
var <список_переменных>;
// Раздел описания заголовков процедур и функций.
procedure <имя>
...
function <имя>
// Далее следует часть реализации модуля - начинается со слова
// implementation.
implementation
// Часть реализации содержит тексты процедур обработки событий,
// а также собственные процедуры и функции программиста.
end.
```

Часть реализации содержит тексты процедур и функций, создаваемых автоматически средой разработки (процедуры обработки событий), а также программистом. В общем случае любая процедура имеет следующую структуру:

```
procedure <имя_процедуры>(<список_параметров>);
label <список_меток>;
const <список_констант>;
type <список_типов>;
var <список_переменных>;
begin
// Операторы процедуры
end;
```

Некоторые разделы в процедуре могут отсутствовать (**label**, **const**, **type**, **var**). Структура функции отличается от процедуры только тем, что имеет в заголовке тип возвращаемого значения:

```
function <имя_функции>(<список_параметров>):<тип_возвращаемого_
значения>;
label <список_меток>;
const <список_констант>;
type <список_типов>;
var <список_переменных>;
begin
// Операторы функции
end;
```

Некоторые разделы в функции также могут отсутствовать (**label**, **const**, **type**, **var**).

Некоторые советы по оформлению программного кода

При оформлении программного кода придерживайтесь следующих правил:

- не следует размещать в одной строке более одного оператора. Это связано с тем, что в случае возникновения ошибок при компиляции не всегда удается точно идентифицировать место ошибки (ошибочный оператор);
- старайтесь как можно чаще использовать комментарии, желательно – по ходу составления программы. Довольно часто разработанные ранее программы приходится модернизировать, поэтому пояснение тех или иных вопросов по ходу текста программы существенно облегчит вам жизнь и сэкономит время разработки (доработки) программы;
- анализируя ту или иную ошибку, иногда полезно посмотреть на оператор, предшествующий тому, что вызвал ошибку. Это также связано с тем, что компилятор далеко не всегда способен точно идентифицировать место ошибки;
- старайтесь выделять тела составных, условных и циклических операторов отступами. Так как зачастую в этих операторах необходимо использовать конструкцию **begin...end**, то без отступов вы можете забыть, какому ключевому слову **begin** соответствует ключевое слово **end**. В результате при компиляции вполне вероятна ситуация, что количество слов **begin** не соответствует количеству слов end. Компилятор в таких случаях выдает два варианта ошибок, означающих избыток или недостаток ключевого слова **end**:
  - `':' expected but '.' found` – количество слов `begin` превышает количество слов `end`;
  - `declaration expected but identifier found` – количество слов `end` превышает количество слов `begin`.
- если тело (текст) вашей программы содержит достаточно большое количество строк, а также содержит при этом относительно самостоятельные части, то попробуйте структурировать свою программу – используйте процедуры и функции.

При использовании идентификаторов старайтесь использовать имена, которые наглядно поясняют назначение того или иного идентификатора. Применение «осмысленных» названий идентификаторов сделает вашу программу более понятной – куда тяжелее разбирать исходный код программы, где присутствует большое количество идентификаторов типа `i`, `j`, `k` и т.п.

Глава 4 Несколько слов об объектно-ориентированном программировании

Эта глава посвящена основам объектно-ориентированного программирования (ООП). Во время написания книги я постарался построить изложение материала таким образом, чтобы для разработки программ в Borland Delphi 2005 знание концепции ООП не являлось необходимым.

Зачем же нужно это самое ООП и действительно ли знать его основы не обязательно? Все дело в том, что ООП – результат накопления опыта программирования, собираемого долгие годы, и в настоящее время представляющее собой мощный инструмент, без которого немыслимо создание ни одной современной программы. Вполне естественно, что понять то, до чего в свое время доходили годами, начинающему программисту будет достаточно сложно.

Именно поэтому материал книги построен таким образом, чтобы у человека, только начинающего программировать, в случае затруднения при усвоении данной главы была бы возможность ее пропустить. В конце концов, немного попрактиковавшись в программировании, вы так или иначе вернетесь к этой главе – еще не зная основ ООП, вы интуитивно ощутите эту необходимость.

Материал данной главы будет весьма полезен для более глубокого понимания того, как программа взаимодействует с компонентами, что и почему среда разработки добавляет в текст программы.

Исторически сложилось так, что программирование возникло и развивалось как *процедурное* программирование. Процедурный подход предполагает, что основой программы является алгоритм, то есть некоторая процедура обработки данных. Вспомните начало второй главы – мы начинали создавать программы, используя именно эти принципы.

Объектно-ориентированное программирование – методика разработки программ, в основе которой лежит понятие «объект». Сразу скажу, что это никоим образом не противоречит тому, что мы делали в самом начале, – на самом деле мы уже тогда при написании программ использовали ООП, просто это было описано в форме, более удобной и доступной для понимания.

Итак, в основе ООП лежит понятие «объект». *Объект* – это некоторая структура, соответствующая объекту реального мира, его поведению. Задача, решаемая с использованием методики ООП, описывается в терминах объектов и операций над ними. Программа при таком подходе представляет собой набор объектов, а также их взаимосвязей. Любой объект принадлежит к какому-либо классу.

Более подробно на понятии «класс» мы остановимся ниже.

Определяем понятие «класс»

Если вспомнить материал прошлой главы, где описываются различные типы данных, то можно сказать, что некоторым аналогом классов являются *записи*. Записи позволяли описать некоторые объекты реального мира (например, сотрудника компании, в структуре записи которого хранилась информации об имени, возрасте и заработной плате). Однако записи позволяли нам описывать только данные об объекте, но не учитывать их поведение.

Язык Delphi, поддерживая концепцию объектно-ориентированного программирования, дает возможность определять классы.

Класс (Class) – это сложная структура, включающая описания не только данных, но и процедур и функций, которые могут быть выполнены над представителем класса (объектом).

Ниже в листинге 4.1 приведен пример объявления класса (как и другие элементы программы – константы, переменные, типы, – классы необходимо объявлять).

Листинг 4.1 ▼ Пример объявления класса

```
// Фрагмент кода, демонстрирующий создание класса.
TPersonal = class
private
  fname:string[30]          // Это свойство класса - имя
                            //сотрудника.
```

```
  fage:byte;                  // Это свойство класса - возраст
                              //сотрудника.
public
  procedure showinfo;         // Это метод класса - процедура,
                              // обеспечивающая показ информации
                              // о сотруднике.
end;
```

Данные класса, описывающие характеристики объекта, называются *полями* (Field) или *атрибутами* (Attribute) , а процедуры и функции, описывающие поведение объекта, – *методами* (Method).

В приведенном примере объявлен класс `TPersonal`. Идентификатор `TPersonal` – это имя класса, `fname` и `fage` – имена полей, `showinfo` – имя метода.

Описание класса помещают в программе в раздел описания типов (`type`).

Что представляет собой объект

После того как класс объявлен, в программе можно создавать (объявлять) *объекты* (Object) – экземпляры класса. Объекты как представители класса объявляются в программе в разделе описания переменных (**var**):

```
var
  boss:TPersonal;
  workers:array [1..100] of TPersonal;
```

В языке Delphi объект – это динамическая структура. Такая структура содержит не сами данные, а ссылку на данные объекта. Поэтому программист должен позаботиться о выделении памяти для этих данных.

Выделение памяти осуществляется при помощи специального метода класса – конструктора, которому обычно присваивают имя `Create` (Создать). Для того чтобы подчеркнуть особую роль и поведение конструктора, в описании класса вместо слова `procedure` используется слово `constructor`. Приведенный ниже в листинге 4.2 фрагмент кода объявляет класс с использованием конструктора.

Листинг 4.2 ▾ Объявление класса, в состав которого введен конструктор

```
// Фрагмент кода, демонстрирующий создание класса с конструктором.
TPersonal = class
private
```

```
    fname:string[30];          // Это свойство класса - имя
                               //сотрудника.
    fage:byte;                 // Это свойство класса - возраст
                               //сотрудника.
  public
    constructor Create;        // Конструктор - процедура создания
                               // нового объекта (экземпляра класса).
    procedure showinfo;        // Это метод класса - процедура,
                               // обеспечивающая показ информации
                               // о сотруднике.
  end;
```

Выделение памяти для данных объекта происходит путем присваивания значения результата применения метода-конструктора к типу (классу) объекта. Например, после выполнения инструкции

```
boss := TPersonal.Create;
```

выделяется необходимая память для данных объекта boss.

Помимо выделения памяти, конструктор, как правило, решает задачу присваивания полям объекта начальных значений, то есть осуществляет *инициализацию* объекта. Ниже приведен пример реализации конструктора для объекта TPersonal:

```
constructor TPerson.Create;
begin
  fname := 'Новый сотрудник';
  fage := 0;
end;
```

После объявления и инициализации объект можно использовать, например установить значение полей объекта. Доступ к полю объекта осуществляется указанием имени объекта и имени поля, которые отделяются друг от друга точкой.

Если в программе какой-либо объект больше не используется, то можно освободить память, занимаемую полями данного объекта. Для выполнения этого действия используют метод Free. Например, для того чтобы освободить память, занимаемую полями объекта boss, достаточно записать:

```
boss.free;
```

На самом деле память освобождается с помощью другого специального метода, называемого деструктором. При его описании используется специальное слово destructor. Метод free всего лишь вызывает деструктор.

Что такое метод

Методы (Method) класса (процедуры и функции, объявление которых включено в описание класса) выполняют некоторые *действия* над объектами класса (то есть характеризуют поведение объектов). Для того чтобы метод был выполнен, необходимо указать имена объекта и метода, отделив одно имя от другого точкой. Например, инструкция

```
boss.showinfo;
```

вызывает применение метода `showinfo` к объекту `boss`. Фактически инструкция применения метода к объекту – это специфический способ записи инструкции вызова процедуры.

Методы класса определяются в программе точно так же, как и обычные процедуры и функции, за исключением того, что имя процедуры или функции, являющейся методом, состоит из двух частей: имени класса, к которому принадлежит метод, и имени метода. Имя класса от имени метода отделяется точкой.

Ниже в листинге 4.3 приведен пример определения метода `showinfo` класса `TPersonal`.

Листинг 4.3. Метод `showinfo` класса `TPersonal`

```
procedure TPersonal.showinfo;
begin
  // Вывод на экран данных о сотруднике.
  messagebox.Show(fname+' '+fage.toString);
end;
```

Как видно из листинга 4.3, в инструкциях метода доступ к полям объекта осуществляется без указания имени объекта.

Далее приведен пример программы (листинг 4.4), в которой объявляется класс `TPersonal`, над объектом которого (`boss`) выполняются различные действия.

Листинг 4.4 Пример программы, демонстрирующий создание и использование класса

```
unit WinForm;
interface
uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data;
```

```
type
  TWinForm = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    Components: System.ComponentModel.Container;
    Button1: System.Windows.Forms.Button;
    procedure InitializeComponent;
    procedure Button1_Click(sender: System.Object;
                            e: System.EventArgs);
  {$ENDREGION}
  strict protected
    procedure Dispose(Disposing: Boolean); override;
  private
    { Private Declarations }
  public
    constructor Create;
  end;
  // Объявление класса.
  TPersonal = class
  private
    fname:string[30];          // Это поле класса - имя сотрудника.
    fage:byte;                 // Это поле класса - возраст
                               // сотрудника.
  public
    constructor Create;        // Конструктор - создание нового
                               // объекта
                               // (экземпляра класса).
    procedure showinfo;        // Это метод класса - показ
                               // информации о
                               // сотруднике.
  end;
  [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm))]
implementation
{$AUTOBOX ON}
{$REGION 'Windows Form Designer generated code'}
procedure TWinForm.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
```

```
  inherited Dispose(Disposing);
end;
// Конструктор для формы.
constructor TWinForm.Create;
begin
  inherited Create;
  InitializeComponent;
end;
// Процедура обработки события Click на кнопке Button1.
procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
var
  worker:TPersonal;
begin
  // Создание нового объекта.
  worker:=TPersonal.Create;
  // Применение метода к объекту - вывод информации о сотруднике.
  worker.showinfo; // На экране сообщение 'Новый сотрудник 0'.
  // Изменяем значение полей объекта.
  worker.fname:='Шупрута Владимир';
  worker.fage:=25;
  // Применение метода к объекту - вывод информации о сотруднике.
  worker.showinfo; // На экране сообщение 'Шупрута Владимир 25'.
  // Уничтожаем объект.
  worker.free;
end;
// Конструктор для класса TPersonal.
// (Создается новый объект с именем 'Новый сотрудник'
//  и возрастом 0)
constructor TPersonal.Create;
begin
  inherited Create;
  fname:='Новый сотрудник';
  fage:=0;
end;
// Метод класса TPersonal - процедура показа данных о сотруднике.
procedure TPersonal.showinfo;
begin
  messagebox.Show(fname+' '+fage.toString);
end;
end.
```

Основные принципы объектно-ориентированного программирования

Мы немного познакомились с классами и объектами. Теперь попробуем разобраться с основными принципами ООП. Вся концепция ООП построена на трех принципах – инкапсуляции, наследовании и полиморфизме. Пугаться столь страшных определений не стоит – далее мы коротко познакомимся с этими принципами, объяснение которых построено на небольшом практическом примере.

Инкапсуляция и свойства объектов

Инкапсуляция (Encapsulation) – это механизм, объединяющий данные и методы их обработки, а также защищающий и то и другое от внешнего вмешательства или неправильного использования. В объектно-ориентированном программировании методы и данные могут быть объединены вместе; в этом случае говорят, что создается так называемый черный ящик.

Внутри объекта методы и данные могут быть закрытыми. Закрытые методы или данные доступны только для других частей этого объекта. Таким образом, закрытые коды и данные недоступны для тех частей программы, которые существуют вне объекта. Если методы и данные являются открытыми, то, несмотря на то что они заданы внутри объекта, они доступны и для других частей программы. Характерной является ситуация, когда открытая часть объекта используется для того, чтобы обеспечить контролируемый интерфейс закрытых элементов объекта.

В языке Delphi ограничение доступа к полям объекта реализуется при помощи *свойств* объекта.

Свойство (Property) объекта – характеристика объекта, которая определяется полем, сохраняющим значение свойства, и двумя методами, обеспечивающими доступ к полю свойства. Метод установки значения свойства называется методом записи свойства (`write`), а метод получения значения свойства – методом чтения свойства (`read`).

В описании класса перед именем свойства записывают слово `property` (свойство). После имени свойства указывается его тип, затем – имена методов, обеспечивающих доступ к значению свойства. После слова `read` указывается имя метода, обеспечивающего чтение свойства, после слова `write` – имя метода, отвечающего за запись свойства. Если не требуется обработка при записи или чтении свойств, то можно просто указать имя поля после ключевых слов `write` и `read`.

Ниже в листинге 4.5 приведен пример описания класса `TPersonal`, содержащего два свойства: `Name` и `Age`.

Листинг 4.5 ▾ Пример описания класса, содержащего свойства Name и Age

```
// Объявление класса.
  TPersonal = class
  type
    TName = string[30];
    TAge = byte;
  private
    fname:TName; // Значение свойства Name - имя сотрудника.
    fage:TAge;   // Значение свойства Age - возраст сотрудника.
    function GetName:TName;
    function Getage:TAge;
    procedure SetAge(new_age:TAge);
  public
    // Конструхтор - создание нового объекта(экземпляра класса).
    constructor Create(Name:TName;Age:TAge);
    procedure showinfo; // Это метод класса - показ информации
                        // о сотруднике.
    // Свойства объекта
    property Name:TName // Свойство только для чтения.
      read GetName;
    property Age:TAge   // Свойство для чтения и записи.
      read GetAge
      write SetAge;
  end;
```

Казалось бы, внешне применение свойств в программе ничем не отличается от использования полей объекта. Однако между свойством и полем объекта существует принципиальное отличие: при присвоении и чтении значения свойства автоматически вызывается процедура (функция), которая выполняет некоторую работу. Например, инструкция

```
worker.Age:=25;
```

на самом деле будет преобразована компилятором в инструкцию вида:

```
worker.SetAge(25);
```

В программе на методы свойства можно возложить некоторые дополнительные задачи. Например, с помощью метода можно проверить корректность присваиваемых свойству значений, установить значения других полей, логически связанных со свойством, вызвать вспомогательную процедуру и т.п.

Оформление данных объекта как свойства позволяет ограничить доступ к полям, хранящим значения свойств объекта, например можно разрешить только чтение. Для того чтобы инструкции программы не могли изменить значение

свойства, в описании свойства надо указать лишь имя метода чтения. Попытка присвоить значение свойству, предназначенному только для чтения, вызывает ошибку времени компиляции.

В приведенном выше описании класса TPersonal свойство Name доступно только для чтения, а свойство Age – для чтения и записи.

Установить значение свойства, защищенного от записи, можно во время инициализации объекта (во время вызова конструктора).

Полный текст программы (с описанием конструктора, процедур и функций) приведен в листинге 4.6.

Листинг 4.6 ▼ Пример программы, демонстрирующей создание и использование класса с описанием свойств

```
unit WinForm;
interface
uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data;
type
  TWinForm = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    Components: System.ComponentModel.Container;
    Button1: System.Windows.Forms.Button;
     procedure InitializeComponent;
     procedure Button1_Click(sender: System.Object;
                             e: System.EventArgs);
  {$ENDREGION}
  strict protected
    procedure Dispose(Disposing: Boolean); override;
  private
    { Private Declarations }
  public
    constructor Create;
  end;
  // Объявление класса TPersonal.
  TPersonal = class
  type
    TName = string[30];
    TAge = byte;
  private
    fname:TName; // Значение свойства Name - имя сотрудника.
    fage:TAge;   // Значение свойства Age - возраст сотрудника.
```

```
    function GetName:TName;
    function GetAge:TAge;
    procedure SetAge(new_age:TAge);
  public
    // Конструктор - создание нового объекта(экземпляра класса).
    constructor Create(Name:TName;Age:TAge);
    procedure ShowInfo;  // Это метод класса - показ информации
                         // о сотруднике.
    // Свойства объекта
    property Name:TName  // Свойство только для чтения.
      read GetName;
    property Age:TAge    // Свойство для чтения и записи.
      read GetAge
      write SetAge;
  end;
  [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm))]
implementation
{$AUTOBOX ON}
{$REGION 'Windows Form Designer generated code'}
procedure TWinForm.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
  inherited Dispose(Disposing);
end;
// Конструктор для формы.
constructor TWinForm.Create;
begin
  inherited Create;
  InitializeComponent;
end;
procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
var
  worker: TPersonal;
begin
  // Создание нового объекта.
  worker:=TPersonal.Create('Шупрута Владимир',0);
  // Вызов метода объекта - вывод информации о сотруднике.
```

```
  worker.showinfo; // На экране сообщение 'Шупрута Владимир 0'.
  // Изменяем свойства объекта.
  worker.age:=25;
  // Вызов метода объекта - вывод информации о сотруднике.
  worker.showinfo; // На экране сообщение 'Шупрута Владимир 25'.
  // Уничтожаем объект.
  worker.free;
end;
// Конструктор для класса TPersonal.
// При вызове создается новый объект с именем Name и возрастом Age.
constructor TPersonal.Create(Name:TName;Age:TAge);
begin
  inherited Create;
  fname:=Name;
  fage:=Age;
end;
// Метод получения значения свойства Name.
function TPersonal.GetName;
begin
  result:=fname;
end;
// Метод получения значения свойства Age.
function TPersonal.GetAge;
begin
  result:=fage;
end;
// Метод записи значения свойства Age.
procedure TPersonal.SetAge(new_age:TAge);
begin
  // Пример обработки значения на запись.
  // Если значение, присваиваемое свойству, <20, то
  // свойство не изменит своего значения.
  if new_age<20
  then exit
  else fage:=new_age;
end;
// Метод класса TPersonal - процедура показа данных о сотруднике.
procedure TPersonal.ShowInfo;
begin
  messagebox.Show(Name+' '+Age.Tostring);
end;
end.
```

Наследование

Концепция объектно-ориентированного программирования предполагает возможность определять новые классы посредством добавления полей, свойств и методов к уже существующим классам. Такой механизм получения новых классов называется порождением. При этом новый, порожденный класс (называемый *потомком*) наследует свойства и методы своего базового, *родительского* класса.

Итак *наследование* (Inheritance) – это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов. Применение иерархии классов делает управляемыми большие потоки информации. Например, представим описание жилого дома. Дом – это часть общего класса, называемого строением. С другой стороны, строение – это часть более общего класса – конструкции, являющейся частью еще более общего класса объектов, который можно назвать созданием рук человека. В каждом случае порожденный класс наследует все связанные с родителем качества и добавляет к ним свои собственные, определяющие, характеристики. Без использования иерархии классов для каждого объекта пришлось бы задать все характеристики, которые бы исчерпывающе его определяли. Однако при использовании наследования можно описать объект путем определения того общего класса (или классов), к которому он относится, с теми специальными чертами, которые делают объект уникальным.

В объявлении класса-потомка указывается класс родителя.

Например, класс `TEmployee` (конкретный сотрудник компании) может быть порожден от рассмотренного выше класса `TPersonal` путем добавления поля `FDepartment` (отдел). Объявление класса `TEmployee` в этом случае может выглядеть так, как показано в листинге 4.7.

Листинг 4.7 ▾ Объявление класса-потомка TEmployee

```
TEmployee = class(TPersonal)
// Класс TEmployee создан на основе TPersonal.
private
  FDepartment:byte; // Номер отдела сотрудника.
  function GetDepartment:byte;
  procedure SetDepartment(New_Department:byte);
public
  constructor Create(Name:TName;Age:TAge;Department:byte);
```

```
  property Department:byte
    read GetDepartment
    write SetDepartment;
end;
```

Заключенное в скобки имя класса TPersonal показывает, что класс TEmployee является производным от класса TPersonal. В свою очередь класс TPersonal является базовым для класса TEmployee.

Класс TEmployee должен иметь свой собственный конструктор, обеспечивающий инициализацию класса-родителя и своих полей. В листинге 4.8 приведен пример реализации конструктора класса TEmployee.

Листинг 4.8 ▼ Конструктор для класса TEmployee

```
// Конструктор для класса TEmployee.
constructor TEmployee.Create(Name:TName;Age:TAge;Department:byte);
begin
  inherited Create(Name,Age); // Инициализация конструктора
                              // класса-родителя.
  FDepartment:=Department;
end;
```

В приведенном примере директивой inherited вызывается конструктор родительского класса. После этого присваивается значение полю класса-потомка.

После создания объекта производного класса в программе можно использовать поля и методы родительского класса. Ниже в листинге 4.9 приведен текст программы, демонстрирующий эту возможность.

Листинг 4.9 ▼ Пример реализации принципа наследования

```
unit WinForm;
interface
uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data;
type
  TName = string[30];
  TAge = byte;
  TWinForm = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    Components: System.ComponentModel.Container;
    Button1: System.Windows.Forms.Button;
```

```
    procedure InitializeComponent;
    procedure Button1_Click(sender: System.Object;
                            e: System.EventArgs);
{$ENDREGION}
strict protected
  procedure Dispose(Disposing: Boolean); override;
private
  { Private Declarations }
public
  constructor Create;
end;
// Объявление класса.
TPersonal = class
private
  fname:TName; // Значение свойства Name - имя сотрудника.
  fage:TAge;   // Значение свойства Age - возраст сотрудника.
  function GetName:TName;
  function GetAge:TAge;
  procedure SetAge(new_age:TAge);
public
  // Конструктор - создание нового объекта (экземпляра класса).
  constructor Create(Name:TName;Age:TAge);
  procedure ShowInfo;  // Это метод класса - показ информации
                       // о сотруднике.
  // Свойства объекта
  property Name:TName  // Свойство только для чтения.
    read GetName;
  property Age:TAge    // Свойство для чтения и записи.
    read GetAge
    write SetAge;
end;
// Класс TEmployee создан на основе TPersonal.
TEmployee = class(TPersonal)
private
  FDepartment:byte; // Номер отдела сотрудника.
  function GetDepartment:byte;
  procedure SetDepartment(New_Department:byte);
public
  constructor Create(Name:TName;Age:TAge;Department:byte);
  property Department:byte
    read GetDepartment
    write SetDepartment;
end;
```

```
[assembly: RuntimeRequiredAttribute(TypeOf(TWinForm))]
implementation
{$AUTOBOX ON}
{$REGION 'Windows Form Designer generated code'}
procedure TWinForm.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
  inherited Dispose(Disposing);
end;
// Конструктор для формы.
constructor TWinForm.Create;
begin
  inherited Create;
  InitializeComponent;
end;
procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
var
  worker: TPersonal;
  dep_worker:TEmployee;
begin
  // Создание нового объекта TPersonal.
  worker:=TPersonal.Create('Шупрута Владимир',0);
  // Вызов метода объекта - вывод информации о сотруднике.
  worker.showinfo;  // На экране сообщение Шупрута Владимир 0'.
                    // Изменяем свойства объекта.
  worker.age:=25;
  // Вызов метода объекта - вывод информации о сотруднике.
  worker.showinfo   // На экране сообщение 'Шупрута Владимир 25'.
                    // Создаем сотрудника - потомка TEmployee.
  dep_worker:=TEmployee.Create('Захаров Сергей',24,1);
  // Отображаем информацию - метод унаследован от TPersonal.
  dep_worker.showinfo; // На экране сообщение 'Захаров Сергей 24'.
                       // Отображаем информацию - поля Name, Age
                       // и Department.
  messagebox.Show(dep_worker.Name + ' '
    + dep_worker.age.tostring+ ' '
    + dep_worker.FDepartment.tostring);
  // Удаляем объекты.
```

```
  worker.free;
  dep_worker.free;
end;
// Конструктор для класса TPersonal.
// При вызове создается новый объект с именем Name и возрастом Age.
constructor TPersonal.Create(Name:TName;Age:TAge);
begin
  inherited Create;
  fname:=Name;
  fage:=Age;
end;
// Метод получения значения свойства Name.
function TPersonal.GetName;
begin
  result:=fname;
end;
// Метод получения значения свойства Age.
function TPersonal.GetAge;
begin
  result:=fage
end;
// Метод записи значения свойства Age.
procedure TPersonal.SetAge(new_age:TAge);
begin
  // Пример обработки значения на запись.
  // Если значение, присваиваемое свойству, <20, то
  // свойство не изменит своего значения.
  if new_age<20
  then exit
  else fage:=new_age;
end;
// Метод класса TPersonal - процедура показа данных о сотруднике.
procedure TPersonal.showinfo;
begin
  messagebox.Show(Name+' '+Age.Tostring);
end;
// Конструктор для класса TEmployee.
constructor TEmployee.Create(Name:TName;Age:TAge;Department:byte);
begin
  inherited Create(Name,Age); // Инициализация конструктора
                              // класса-родителя.
  FDepartment:=Department;
```

```
end;
// Метод получения значения свойства Department.
function TEmployee.GetDepartment;
begin
  result:=FDepartment;
end;
// Метод записи значения в свойство Department.
procedure TEmployee.SetDepartment(New_Department:byte);
begin
  FDepartment:=New_Department;
end;
end.
```

Несколько слов о директиве **inherited**. Данная директива «говорит» компилятору, что нужно вызвать метод базового (родительского) класса с тем же именем, что и метод, в котором она указана. При этом если сигнатура методов родительского класса и его потомка совпадают (напомню, что сигнатурой метода является его имя и набор параметров), то указывать полный формат вызова не требуется. Например, создадим потомка класса TEmployee, который отличается от своего родителя только тем, что при вызове конструктора сразу выводится сообщение с информацией из полей объекта:

```
TNewEmployee = class(TEmployee)
public
  constructor Create(Name:TName;Age:TAge;Department:byte);
end;
constructor TNewEmployee.Create(Name:TName;Age:TAge;Department:byte);
begin
  inherited;
  ShowInfo;
end;
```

Как видите, в конструкторе просто указывается директива **inherited**, а компилятор сам подставит за ней вызов конструктора родительского объекта с правильными параметрами.

Однако при внимательном рассмотрении листинга 4.9, а точнее – описания объекта TPersonal и реализации его конструктора – у вас может возникнуть следующий вопрос: как же так, в описании объекта отсутствует указание класса-родителя, а в конструкторе мы обращаемся к конструктору базового класса? Дело в том, что в Delphi все классы выстроены в единую иерархию, на вершине которой стоит класс TObject. Когда при описании класса пропускается указание его родителя, то в этом случае компилятор считает, что родителем такого класса является TObject.

Зачем нужны директивы protected и private

Помимо объявления элементов класса (полей, методов, свойств) описание класса, как правило, содержит директивы `protected` (защищенный) и `private` (частный), которые устанавливают *степень видимости* элементов класса в программе.

Элементы класса, объявленные в секции `protected`, доступны только в порожденных от него классах. Область видимости элементов класса этой секции не ограничивается модулем, в котором находится описание класса. Обычно в секцию `protected` помещают описание методов класса.

Элементы класса, объявленные в секции `private`, видимы только внутри модуля. Эти элементы не доступны за пределами модуля, даже в производных классах. Обычно в секцию `private` помещают описание полей класса, а методы, обеспечивающие доступ к этим полям, помещают в секцию `protected`.

В тех случаях, когда нужно полностью скрыть элементы класса, определение класса следует поместить в отдельный модуль, а в программу, которая использует объекты этого класса, поместить в секции `uses` ссылку на этот модуль.

Полиморфизм

Из прошлого примера вы, вероятно, заметили одну важную особенность. Класс TEmployee наследует метод ShowInfo класса TPersonal, но данный метод для класса TEmployee не очень полезен, так как отображает только часть информации – имя и возраст. Конечно, хотелось бы, чтобы при использовании данного метода для класса TEmployee выводилась полная информация – имя, возраст и отдел. Такая замена методов при отсутствии внешних различий в вызовах может быть реализована с помощью третьей концепции ООП – полиморфизма.

Полиморфизм (Polymorphism) – это возможность использовать одинаковые имена для методов, входящих в различные классы. Концепция полиморфизма обеспечивает в случае применения метода к объекту использование именно того метода, который соответствует классу объекта.

Из прошлого примера у нас определены два класса: `TPersonal` и `TEmployee`, причем первый является базовым для второго.

В базовом классе определен метод `ShowInfo`, обеспечивающий вывод информации (имени и возраста) о сотруднике на экран. Чтобы дочерний класс (потомок) мог использовать метод с таким же именем, суть которого составляли бы несколько другие действия, данный метод в базовом классе стоит объявить с директивой `virtual`. Объявление метода виртуальным дает возможность дочернему классу произвести замену виртуального метода своим

собственным. Ниже в листинге 4.10 приведен пример описания базового класса с использованием директивы virtual.

Листинг 4.10 ▼ Метод ShowInfo объявлен в базовом классе как виртуальный

```
// Объявление класса.
  TPersonal = class
  private
    fname:TName;     // Значение свойства Name - имя сотрудника.
    fage:TAge;       // Значение свойства Age - возраст сотрудника.
    function GetName:TName;
    function GetAge:TAge;
    procedure SetAge(new_age:TAge);
  public
    // Конструктор - создание нового (экземпляра класса).
    constructor Create(Name:TName;Age:TAge);
   // Это метод класса - показ информации о сотруднике - виртуальный.
    procedure ShowInfo; virtual;
    // Свойства объекта
    property Name:TName  // Свойство только для чтения.
      read GetName;
    property Age:TAge    // Свойство для чтения и записи.
      read GetAge
      write SetAge;
  end;
```

В дочернем классе TEmployee также определен свой метод ShowInfo (листинг 4.11), который замещает соответствующий метод родительского класса. Метод порожденного класса, замещающий виртуальный метод родительского класса, помечается директивой override.

Листинг 4.11 ▼ Метод ShowInfo в описании класса TEmployee помечен директивой override

```
// Класс TEmployee создан на основе TPersonal.
TEmployee = class(TPersonal)
  FDepartment:byte; // Номер отдела сотрудника.
  function GetDepartment:byte;
  procedure SetDepartment(New_Department:byte);
public
  constructor Create(Name:TName;Age:TAge;Department:byte);
  procedure showinfo; override; // Метод ShowInfo заменен
                                // на собственный.
  // Свойства объекта.
  property Department:byte
```

```
    read GetDepartment
    write SetDepartment;
end;
```

Несколько слов о классах и объектах Delphi

Кратко рассмотрев основные принципы ООП, можно немного по-другому взглянуть на исходный код программ, написанных на языке Delphi. После создания нового проекта (например, проекта для платформы Win32) исходный код выглядит так, как показано в листинге 4.12.

Листинг 4.12 ▾ Заготовка исходного кода для нового проекта Win32

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

Для реализации интерфейса будущей программы среда разработки Borland Delphi 2005 использует библиотеку классов, содержащую большое количество разнообразных классов. С помощью этих классов описывается форма приложения, а также различные компоненты формы (командные кнопки, поля редактирования, метки и т.д.).

Во время проектирования формы приложения среда разработки автоматически добавляет в текст программы необходимые объекты. Когда программист, добавляя необходимые компоненты, формирует будущее диалоговое окно, Delphi формирует описание класса формы. Когда программист создает функцию обработки события формы или ее компонента, Delphi добавляет объявление метода в описание класса формы приложения.

В листинге 4.13 приведен измененный исходный код проекта после добавления на форму компонента-кнопки (TButton) и описания функции обработки события Click этой кнопки.

Листинг 4.13 ▼ Теперь в описании класса формы произошли изменения

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
// Объявление класса формы.
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject); // Объявление метода.
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
// Реализация метода.
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Обработка события OnClick');
end;
end.
```

Помимо классов визуальных компонентов в библиотеку классов входят и классы так называемых *невизуальных* (невидимых) компонентов, которые обеспечивают создание соответствующих объектов и доступ к их методам и свойствам. Типичным примером невизуального компонента является таймер (TTimer).

На этом мы заканчиваем теоретическое знакомство с принципами объектно-ориентированного программирования. Следующая глава будет посвящена рассмотрению основных компонентов, с помощью которых создаются программы для операционной системы Windows.

Глава 5

Изучаем основные компоненты при программировании для .NET

В этой главе приведено описание базовых компонентов. Следует обратить внимание, что показать назначение и возможности какого-либо компонента практически невозможно, не используя другие компоненты. Поэтому в приведенных примерах основное внимание уделяется изучаемому компоненту, название которого вынесено в заголовок раздела. В таблицах приводятся основные свойства, которые отражают специфику компонента и представляют для начинающего программиста наибольший интерес. Более подробную информацию о компонентах вы можете найти в справочной системе.

Базовые компоненты располагаются на вкладках **Windows Forms** и **Components** окна **Tool Palette**. Некоторые компоненты нам уже знакомы – они кратко будут рассмотрены вначале, на незнакомых компонентах мы остановимся более подробно.

Компонент Label

С этим компонентом мы уже успели познакомиться, когда создавали наш первый проект. Как вы уже знаете, этот компонент предназначен для отображения текстовой информации. Текст, который будет отображен, можно задавать как на этапе разработки формы, так и в процессе выполнения программы, присвоив значение свойству `Text`. Основные свойства компонента приведены в табл. 5.1.

Таблица 5.1 ▼ Основные свойства компонента `Label`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Text`	Текст, отображаемый в поле компонента
`Font`	Шрифт, который используется для отображения текста
`ForeColor`	Цвет текста, находящегося в поле компонента
`BackColor`	Цвет фона поля компонента
`Location.X`	Расстояние от левой границы формы до левой границы компонента
`Location.Y`	Расстояние от верхней границы формы до верхней границы компонента
`Size.Width`	Ширина поля компонента
`Size.Height`	Высота поля компонента
`BorderStyle`	Вид рамки компонента. По умолчанию рамка отсутствует (`None`). Свойство также может принимать значения `FixedSingle` (Тонкая рамка) и `Fixed3D` (Объемная рамка)
`TextAlign`	Способ выравнивания текста в поле компонента. Текст может быть прижат к левому краю по высоте (`TopLeft`), посередине (`TopCentre`) или быть выровненным по центру (`MiddleCenter`)

Чтобы в поле компонента `Label` вывести числовое значение, это значение необходимо при помощи функции `ToString` преобразовать в строку.

Цвет текста (`ForeColor`) и фона (`BackColor`) можно задать, указав название цвета (`Green`, `Red`, `Blue` и т.п.) или элемент цветовой схемы операционной системы (например, `System.Drawing.SystemColors.Control`). В чем разница? Она состоит в том, что цветовая схема вашего приложения во втором случае будет привязана к цветовой схеме операционной системы и будет автоматически меняться при каждой ее смене. По умолчанию используется как раз второй вариант. Цвет фона также может быть и прозрачным (`Color.Transparent`).

Теперь рассмотрим небольшую программу, демонстрирующую нам основные свойства этого компонента в действии. На рис. 5.1 приведен внешний вид окна программы, а в листинге 5.1 – процедуры обработки события `Click` на соответствующих кнопках.

Листинг 5.1 ▼ Текст процедур обработки событий `Click` на кнопках

```
procedure TWinForm.RadioButton6_CheckedChanged(sender: System.Object;
                                               e: System.EventArgs);
begin
 Label1.TextAlign := ContentAlignment.TopRight;
end;

procedure TWinForm.RadioButton5_CheckedChanged(sender: System.Object;
                                               e: System.EventArgs);
```

```
begin
 Label1.TextAlign := ContentAlignment.TopCenter;
end;

procedure TWinForm.RadioButton4_CheckedChanged(sender: System.Object;
                                               e: System.EventArgs);
begin
 Label1.TextAlign := ContentAlignment.TopLeft;
end;

procedure TWinForm.RadioButton1_CheckedChanged(sender: System.Object;
                                               e: System.EventArgs);
begin
  Label1.BorderStyle := BorderStyle.None;
end;

procedure TWinForm.RadioButton3_CheckedChanged(sender: System.Object;
                                               e: System.EventArgs);
begin
 Label1.BorderStyle := BorderStyle.Fixed3D;
end;

procedure TWinForm.RadioButton2_CheckedChanged(sender: System.Object;
                                               e: System.EventArgs);
begin
 Label1.BorderStyle := BorderStyle.FixedSingle;
end;
```

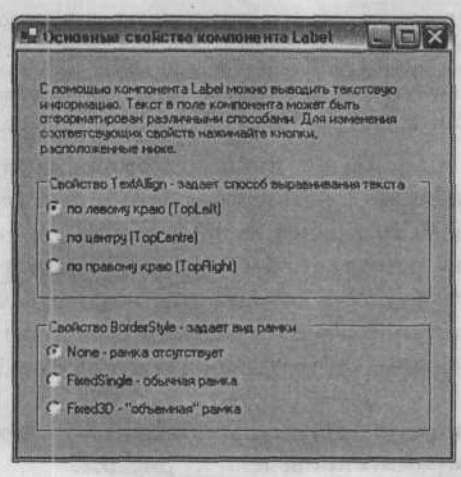

Рис. 5.1 ▾ Окно программы, демонстрирующей основные свойства компонента Label

Компонент TextBox

Это тоже знакомый нам компонент. С его помощью мы вводили данные с клавиатуры. Ниже в табл. 5.2 приведены основные свойства этого компонента.

Таблица 5.2 ▼ Основные свойства компонента `TextBox`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Text`	Текст, отображаемый в поле компонента
`MaxLength`	Максимально допустимое число символов, которое можно ввести в поле компонента
`Multiline`	Разрешает (`Multiline=True`) или запрещает (`Multiline=False`) ввод нескольких строк текста
`Lines`	Массив строк, элементами которого являются строки, расположенные в поле редактирования в случае использования режима `Multiline`
`Font`	Шрифт, который используется для отображения текста
`ForeColor`	Цвет текста, находящегося в поле компонента
`BackColor`	Цвет фона поля компонента
`Location.X`	Расстояние от левой границы формы до левой границы компонента
`Location.Y`	Расстояние от верхней границы формы до верхней границы компонента
`Size.Width`	Ширина поля компонента
`Size.Height`	Высота поля компонента
`BorderStyle`	Вид рамки компонента. Свойство может принимать значения `FixedSingle` (Тонкая рамка) и `Fixed3D` (Объемная рамка) и `None` (Рамка отсутствует)
`TextAlign`	Способ выравнивания текста в поле компонента. Текст может быть прижат к левому краю (`Left`), правому краю (`Right`) или быть выровненным по центру (`Center`)
`ScrollBars`	Свойство задает вид полосок прокрутки. Может принимать значения `Horizontal` (горизонтальные полосы прокрутки), `Vertical` (вертикальные), `Both` (задействовать оба вида), `None` (не отображать полосы прокрутки)

По умолчанию в поле этого компонента отображаются *все* символы, которые пользователь набирает с клавиатуры. Это может быть не всегда удобно, поэтому предусмотрена возможность фильтрации вводимых символов. В качестве примера скажу, что для того чтобы отображать только нужные символы, необходимо для данного компонента написать обработчик события `KeyPress` (возникает при нажатии клавиши). Если вы не хотите, чтобы какой-либо символ отображался, то присвойте в поле `Handled` параметра `e` значение `True`.

Ниже приведен пример программы (листинг 5.2), поясняющий все вышесказанное. Окно программы приведено на рис. 5.2.

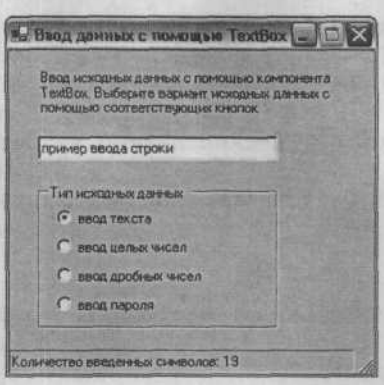

Рис. 5.2 ▾ Окно программы, демонстрирующей особенности использования компонента `TextBox`

Листинг 5.2 ▾ Текст программы, демонстрирующей использование компонента `TextBox` для ввода различных данных

```
unit WinForm1;

interface

uses
 System.Drawing, System.Collections, System.ComponentModel,
 System.Windows.Forms, System.Data, System.Globalization;

type
 TWinForm1 = class(System.Windows.Forms.Form)
 {$REGION 'Designer Managed Code'}
 strict private
  Components: System.ComponentModel.Container;
  Label1: System.Windows.Forms.Label;
  RadioButton1: System.Windows.Forms.RadioButton;
  RadioButton2: System.Windows.Forms.RadioButton;
  RadioButton3: System.Windows.Forms.RadioButton;
  RadioButton4: System.Windows.Forms.RadioButton;
  TextBox1: System.Windows.Forms.TextBox;
  StatusBar1: System.Windows.Forms.StatusBar;
  GroupBox1: System.Windows.Forms.GroupBox;
  procedure InitializeComponent;
  procedure TextBox1_KeyPress(sender: System.Object;
                        e: System.Windows.Forms.KeyPressEventArgs);
  procedure RadioButton1_CheckedChanged(sender: System.Object;
                                        e: System.EventArgs);
```

```
  procedure RadioButton4_CheckedChanged(sender: System.Object;
                                        e: System.EventArgs);
  procedure RadioButton3_CheckedChanged(sender: System.Object;
                                        e: System.EventArgs);
  procedure RadioButton2_CheckedChanged(sender: System.Object;
                                        e: System.EventArgs);
  procedure TextBox1_TextChanged(sender: System.Object;
                                 e: System.EventArgs);
{$ENDREGION}
strict protected
  procedure Dispose(Disposing: Boolean); override;
private
  {Private Declarations}
  input_ch:integer; // Переменная для указания типа данных,
           // которые можно вводить в TextBox1.
           // Обозначим числами следующие типы данных:
           // 0 - строка символов;
           // 1 - целое число;
           // 2 - дробное число;
           // 3 - пароль.
  ds:Char;
public
  constructor Create;
end;

[assembly: RuntimeRequiredAttribute(TypeOf(TWinForm1))]

implementation

{$AUTOBOX ON}

procedure TWinForm1.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
  inherited Dispose(Disposing);
end;

// Процедура, выполняемая в момент создания окна приложения.
constructor TWinForm1.Create;
begin
  inherited Create;
  InitializeComponent;
```

```
 input_ch:=0;
 ds := NumberFormatInfo.CurrentInfo.NumberDecimalSeparator[1];
 StatusBar1.Panels.Add('0');
 StatusBar1.Panels[0].AutoSize := StatusBarPanelAutoSize.Spring;
end;

// Процедура определения нажатой клавиши (отображаемого символа).
procedure TWinForm1.TextBox1_KeyPress(sender: System.Object;
                        e: System.Windows.Forms.KeyPressEventArgs);
begin
 if (e.KeyChar = Chr(8)) then exit;
 case input_ch of
 0:; // Ввод строки текста - ограничений по вводу нет.
 1: // Ввод целых чисел.
   case e.KeyChar of
    '0' .. '9' : ;
    '-': if TextBox1.Text.IndexOf('-') <> -1
         then e.Handled := True;
    else e.Handled := True;
   end;
 2: // Ввод дробных чисел.
   case e.KeyChar of
    '0' .. '9' : ;
    '-': if TextBox1.Text.IndexOf('-') <> -1
         then e.Handled := True;
    else
       if (e.KeyChar =
            NumberFormatInfo.CurrentInfo.NumberDecimalSeparator[1])
         then
          begin
           if (TextBox1.Text.IndexOf(e.KeyChar) <> -1)
            then e.Handled := True;
          end
         else e.Handled := True;
   end;
 3: // Ввод пароля (отображается звездочками).
   case e.KeyChar of
    '0'..'9','A'..'z','A'..'я':
       if TextBox1.Text.Length = 12
         then e.Handled := True;
    else e.Handled := True;
   end;
 end;
end;
```

```
// Процедура отображения количества введенных символов
// вызывается в случае, если изменилось содержимое поля ввода
// TextBox1.
procedure TWinForm1.TextBox1_TextChanged(sender: System.Object; e:
System.EventArgs);
begin
  StatusBar1.Panels[0].Text := 'Количество введенных символов:
'+Convert.ToString(TextBox1.Text.Length);
end;
// Установка режима ввода текста (при нажатии на кнопку
// RadioButton1).
procedure TWinForm1.RadioButton1_CheckedChanged(sender: System.Object;
                                                e: System.EventArgs);
begin
  TextBox1.Clear;          // Очистка поля для ввода.
  TextBox1.Select;         // Курсор устанавливается в поле
                           // компонента.
  TextBox1.MaxLength:= 20;   // Ограничение максимального числа
                             // вводимых символов.
  TextBox1.PasswordChar:=Chr(0); // Режим ввода пароля отключен.
  input_ch:=0;
end;

// Установка режима ввода целых чисел (при нажатии на кнопку
// RadioButton2).
procedure TWinForm1.RadioButton2_CheckedChanged(sender: System.Object;
                                                e: System.EventArgs);
begin
  TextBox1.Clear;
  TextBox1.Select;
  TextBox1.MaxLength:=9;
  TextBox1.PasswordChar:=Chr(0);
  input_ch:=1;
end;

// Установка режима ввода дробных чисел (при нажатии на кнопку
// RadioButton3).
procedure TWinForm1.RadioButton3_CheckedChanged(sender: System.Object;
                                                e: System.EventArgs);
begin
  TextBox1.Clear;
  TextBox1.Select;
  TextBox1.MaxLength:=12;
  TextBox1.PasswordChar:=Chr(0);
  input_ch:=2;
```

```
end;

// Установка режима ввода пароля (сиволов звездочки) (при нажатии
// на кнопку RadioButton4).
procedure TWinForm1.RadioButton4_CheckedChanged(sender: System.Object;
                                                e: System.EventArgs);
begin
 TextBox1.Clear;
 TextBox1.Select;
 TextBox1.MaxLength:=12;
 TextBox1.PasswordChar:='*';
 input_ch:=3;
end;
end.
```

Компонент Button

Это последний из рассмотренных нами ранее компонентов. Компонент `Button` представляет собой командную кнопку. Свойства компонента приведены в табл. 5.3.

Таблица 5.3 ▾ Основные свойства компонента `Button`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Text`	Текст, отображаемый на кнопке
`Font`	Шрифт, который используется для отображения текста
`ForeColor`	Цвет текста, отображаемого на кнопке
`Location.X`	Расстояние от левой границы формы до левой границы компонента
`Location.Y`	Расстояние от верхней границы формы до верхней границы компонента
`Size.Width`	Ширина поля компонента
`Size.Height`	Высота поля компонента
`Enabled`	Признак доступности кнопки. Кнопка доступна, если значение свойства равно `True`, если же оно равно `False` – кнопка недоступна
`Visible`	Признак видимости кнопки на поверхности формы. Если значение свойства равно `True` – кнопка отображается, в противном случае – кнопка невидима
`TextAlign`	Способ выравнивания текста в поле компонента. Текст может быть прижат различными способами – `TopLeft`, `TopCenter`, `TopRight`, `MiddleLeft`, `MiddleCenter`, `MiddleRight`, `BottomLeft`, `BottomCenter`, `BottmRight`
`Image`	Картинка на кнопке. Можно использовать различные форматы – bmp, gif, jpg и т.д. Рекомендуется использовать формат gif, позволяющий задавать прозрачный цвет для картинки
`ImageAlign`	Способ выравнивания картинки на кнопке. Картинка может быть прижата к левой границе кнопки (`MiddleLeft`), правой (`MiddleRight`), располагаться в центре (`MiddleCenter`). Также возможны и другие варианты, аналогичные используемым в свойстве `TextAlign`

Таблица 5.3 ▼ Основные свойства компонента `Button` (окончание)

Свойство	Комментарий
`ImageList`	Свойство определяет используемый компонент `ImageList` (набор картинок) в качестве источника картинок для обозначения различных состояний кнопки. Представляет собой ссылку на объект типа `ImageList` (см. ниже). Чтобы задать свойство, требуется добавить компонент `ImageList` на форму
`ImageIndex`	Номер (индекс) картинки из набора компонента `ImageList`, которая отображается на кнопке

Как видно из табл. 5.3, кнопка может содержать картинку. Добавить ее можно двумя способами – указать значение свойства `Image` либо добавить на форму компонент-контейнер картинок `ImageList` (описание компонента см. ниже) и установить связь между этими компонентами. Первый способ значительно проще, однако таким образом нельзя задавать прозрачный цвет, поэтому цвет фона должен совпадать с цветом кнопки.

Компонент ImageList

Компонент `ImageList` представляет собой контейнер, содержащий набор картинок. Эти картинки могут быть использованы другими компонентами (например, компонентами `Button` или `ToolBar`). Компонент не отображается в процессе выполнения программы, то есть является *невизуальным*. Если вы попробуете добавить компонент `ImageList` в проект, то он будет перенесен не на форму, а в нижнюю часть окна **Design** (рис. 5.3). Точно также отображаются и другие невизуальные компоненты – в нижней части окна **Design**.

Основные свойства этого компонента приведены ниже в табл. 5.4.

Таблица 5.4 ▼ Основные свойства компонента `ImageList`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Images`	Коллекция картинок
`ImageSize.Width`	Ширина картинок коллекции
`ImageSize.Height`	Высота картинок коллекции
`ColorDepth`	Глубина цвета (количество байтов, используемых для кодирования цвета пикселя)
`TransparentColor`	Свойство задает прозрачный цвет. Точки, имеющие этот цвет, не отображаются

Набор картинок формируется во время разработки формы из заранее подготовленных картинок. Формат исходных картинок может быть практически любым (bmp, gif, png, ico). Все картинки должны быть *одного* размера и иметь

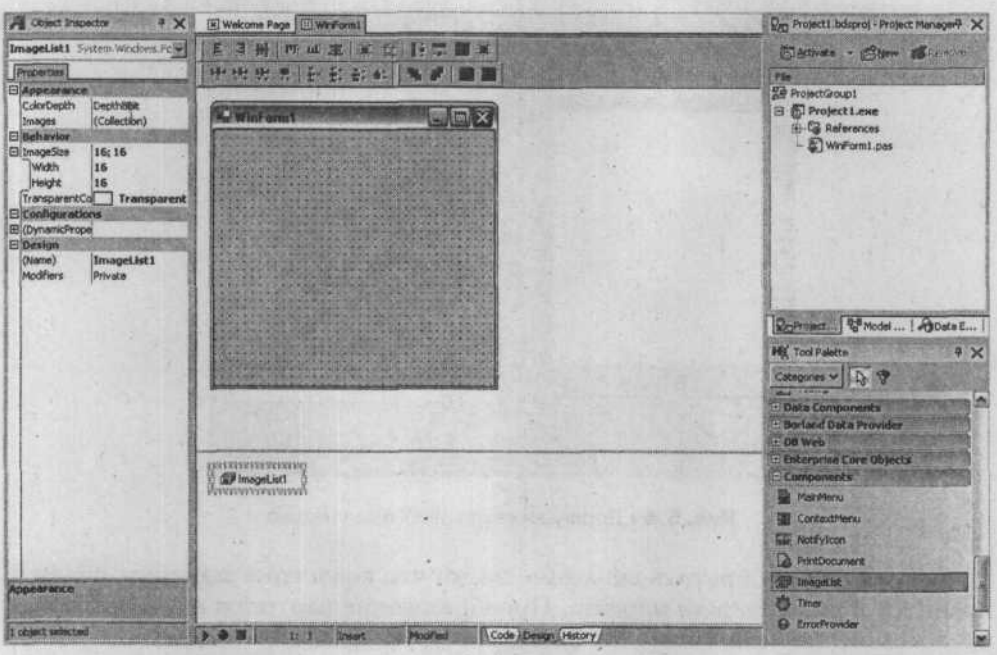

Рис. 5.3 ▾ Невизуальные компоненты отображаются в нижней части окна **Design**

одинаковый цвет фона. Формируется коллекция путем добавления в нее элементов. Чтобы добавить элемент в коллекцию картинок, следует в строке свойства **Images** щелкнуть по кнопке с тремя точками. На экране появится окно **Image Collection Editor** – окно редактора коллекции (рис. 5.4). В этом окне щелкните по кнопке **Add** и в появившемся стандартном окне открытия файла выберите файл картинки.

После того как коллекция будет сформирована, необходимо:

1. Задать размер картинок коллекции (путем присвоения значения свойству `ImageSize`).
2. Определить прозрачный цвет (присвоив значение свойству `TransparentColor`).
3. Задать глубину цветовой палитры (присвоив значение свойству `ColorDepth`).

При задании параметров коллекции имейте в виду, что если размер картинок коллекции не совпадает с выбранным значением свойства `ImageSize`, то будет произведено масштабирование.

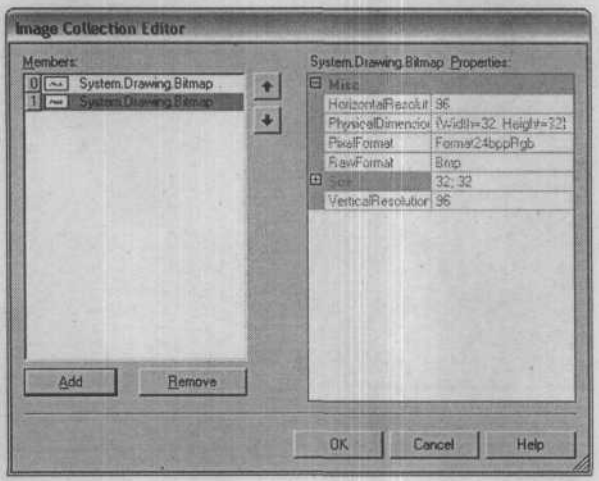

Рис. 5.4 ▼ Добавление картинок в коллекцию

Следует также обратить внимание на то, что коллекция картинок объекта хранится в *файле ресурсов* проекта. Преобразование картинок осуществляется редактором свойств **Image Collection Editor**, при этом исходные файлы картинок для работы программы не нужны.

Компонент ToolTip

Данный компонент является вспомогательным. Основное его назначение состоит в предоставлении другим компонентам сведений-подсказок, которые появляются при наведении указателя мыши на эти компоненты. Основные свойства компонента приведены в табл. 5.5.

Таблица 5.5 ▼ Основные свойства компонента `ToolTip`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
`Active`	Свойство разрешает (`Active=True`) или запрещает (`Active=False`) отображение подсказок
`AutoPopDelay`	Время отображения подсказки
`InitialDelay`	Время, в течение которого указатель мыши должен оставаться неподвижным, чтобы появилась подсказка
`ReshowDelay`	Время задержки отображения подсказки после перемещения указателя мыши с одного компонента на другой

После того как компонент будет добавлен в форму, у других компонентов становится доступным свойство ToolTip, которое и определяет текст подсказки.

Компонент Panel

Компонент Panel представляет собой контейнер для других компонентов и позволяет легко управлять компонентами, которые находятся на панели. Смысл состоит в том, что компоненты, находящиеся на панели, *наследуют* свойства компонента Panel. Например, чтобы сделать недоступными все компоненты на панели, достаточно присвоить значение False свойству Enabled панели. Свойства компонента Panel приведены в табл. 5.6.

Таблица 5.6 ▾ Основные свойства компонента Panel

Свойство	Комментарий
Name	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
BorderStyle	Вид границы панели – обычная рамка (FixedSingle), «объемная» граница (Fixed3D), нет границы (None)
BackGroundImage	Позволяет задать фоновый рисунок панели. Возможно использование форматов bmp, jpg, png, gif и ico
Enabled	Свойство позволяет сделать доступными (Enabled=True) или недоступными (Enabled=False) все компоненты, которые размещены на панели
Visible	Свойство позволяет отображать (Visible=True) и скрывать (Visible=False) панель
Dock	Определяет границу формы, к которой «прикреплена» панель. Панель может быть прикреплена к верхней (Top), нижней (Bottom), левой (Left) или правой (Right) границе формы, занимать всю форму (Fill) либо быть независимой (None)
Font	Задает шрифт панели. Все элементы, размещенные на панели, будут иметь указанный шрифт
AutoScroll	Признак необходимости отображать (AutoScroll=True) полосы прокрутки в случае, если компоненты, находящиеся на панели, не могут быть выведены полностью

Из перечисленных в табл. 5.6 свойств более подробно следует остановиться на свойстве Dock. Это свойство позволяет «прикрепить» панель к границе формы. В результате привязки панели к границе формы размер панели автоматически меняется.

Ниже на рис. 5.5 приведено окно программы, демонстрирующей некоторые особенности использования компонента Panel.

На панель помещены различные компоненты. Все они наследуют свойства панели, на которой располагаются. Программа позволяет изменить шрифт панели, а также сделать ее доступной или недоступной. Соответственно,

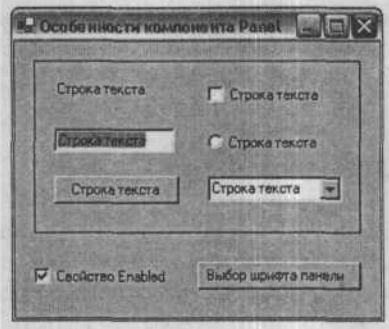

Рис. 5.5 ▼ Использование компонента `Panel`

изменяется шрифт и доступность компонентов, расположенных на панели. Особенности использования компонента `Panel` показаны в листинге 5.3.

Листинг 5.3 ▼ Фрагмент программы, демонстрирующей особенности использования компонента `Panel`

```
//Свойство Enabled панели.
procedure TWinForm1.CheckBox1_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 Panel1.Enabled:=CheckBox1.Checked;
end;

//Свойство Font панели.
procedure TWinForm1.Button2_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
 FontDialog1.ShowDialog;
 Panel1.Font:=FontDialog1.Font;
end;
```

Компонент CheckBox

Компонент `CheckBox` является переключателем, который может находиться в одном из двух состояний: выбранном или невыбранном (иногда еще говорят установленном или не установленном). Рядом с переключателем обычно находится поясняющий текст.

Свойства компонента `CheckBox` приведены в табл. 5.7. Состояние переключателя изменяется в результате щелчка по его изображению (если значение свойства `AutoCheck` равно `True`). При этом возникает событие

CheckedChanged, потом – событие Click. Если же значение свойства AutoCheck равно False, то в результате щелчка на переключателе возникает событие Click, а затем, если процедура обработки этого события изменит состоянии кнопки, возникает событие CheckedChanged.

Таблица 5.7 ▾ Основные свойства компонента CheckBox

Свойство	Комментарий
Name	Имя компонента. Это имя используется для доступа в программе к компоненту и его свойствам
Text	Текст, располагающийся справа от флажка
Checked	Свойство определяет, в каком состоянии находится переключатель. Если переключатель выбран, то значение свойства равно True, если не выбран – False
Enabled	Свойство, определяющее, доступен ли переключатель (Enabled=True) или нет (Enabled=False)
Visible	Свойство позволяет отображать (Visible=True) и скрывать (Visible=False) панель
CheckAlign	Положение кнопки в поле компонента
TextAlign	Положение текста в поле отображения текста. Свойство может принимать следующие значения: MiddleLeft, MiddleCenter, MiddleRight, TopLeft, TopCenter, TopRight, BottomLeft, BottomCenter, BottomRight
Image	Картинка, отображаемая в поле компонента
ImageAlign	Положение картинки в поле компонента. Свойство может принимать следующие значения: MiddleLeft, MiddleCenter, MiddleRight, TopLeft, TopCenter, TopRight, BottomLeft, BottomCenter, BottomRight
BackgroundImage	Картинка, определяющая фон компонента
ImageList	Набор картинок, используемых для обозначения различных состояний кнопки. Свойство содержит ссылку (имя) на объект типа ImageList
ImageIndex	Номер (индекс) картинки из набора ImageList
Appearance	Свойство определят вид переключателя. Переключатель может быть обычным (Normal) или выполненным в виде кнопки (Button)
AutoCheck	Свойство определяет, будет ли автоматически изменяться состояние переключателя в результате щелчка по его изображению. По умолчанию значение этого свойства установлено в True
FlatStyle	Задает стиль переключателя. Переключатель может быть стандартным (Normal), плоским (Flat) или «всплывающим» (Popup). Стиль отображения проявляется при наведении на переключатель указателя мыши
ThreeState	Свойство определяет, будет ли иметь переключатель три состояния. По умолчанию значение свойства установлено в False

Следующая программа (ее форма приведена на рис. 5.6, а текст – в листинге 5.4) демонстрирует использование компонента CheckBox. Программа позволяет вычислить стоимость заказа в кафе.

Листинг 5.4 ▾ Текст программы вычисления стоимости заказа

```
unit WinForm1;

interface
```

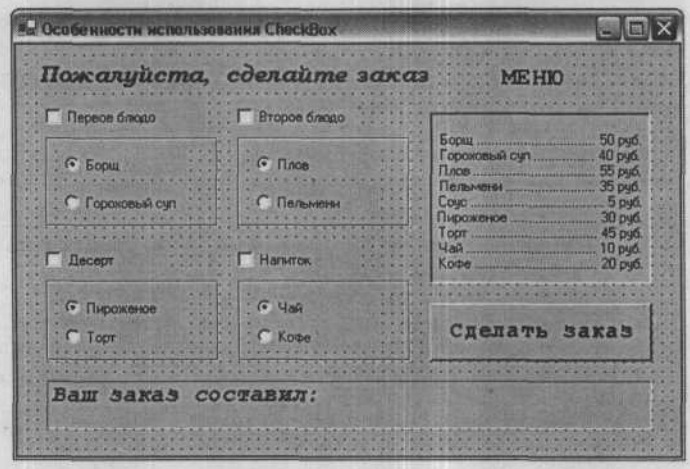

Рис. 5.6 ▼ Форма окна программы вычисления стоимости заказа

```
uses
 System.Drawing, System.Collections, System.ComponentModel,
 System.Windows.Forms, System.Data;

type
 TWinForm1 = class(System.Windows.Forms.Form)
 {$REGION 'Designer Managed Code'}
 strict private
  Components: System.ComponentModel.Container;
  Label1: System.Windows.Forms.Label;
  CheckBox1: System.Windows.Forms.CheckBox;
  CheckBox2: System.Windows.Forms.CheckBox;
  CheckBox3: System.Windows.Forms.CheckBox;
  CheckBox4: System.Windows.Forms.CheckBox;
  Label2: System.Windows.Forms.Label;
  GroupBox1: System.Windows.Forms.GroupBox;
  GroupBox2: System.Windows.Forms.GroupBox;
  GroupBox3: System.Windows.Forms.GroupBox;
  GroupBox4: System.Windows.Forms.GroupBox;
  RadioButton1: System.Windows.Forms.RadioButton;
  RadioButton2: System.Windows.Forms.RadioButton;
  RadioButton3: System.Windows.Forms.RadioButton;
  RadioButton4: System.Windows.Forms.RadioButton;
  RadioButton5: System.Windows.Forms.RadioButton;
  RadioButton6: System.Windows.Forms.RadioButton;
  RadioButton7: System.Windows.Forms.RadioButton;
```

```
  RadioButton8: System.Windows.Forms.RadioButton;
  TextBox1: System.Windows.Forms.TextBox;
  Label3: System.Windows.Forms.Label;
  Button1: System.Windows.Forms.Button;
  procedure InitializeComponent;
  procedure CheckBox1_Click(sender: System.Object;
                            e: System.EventArgs);
  procedure CheckBox2_Click(sender: System.Object;
                            e: System.EventArgs);
  procedure CheckBox3_Click(sender: System.Object;
                            e: System.EventArgs);
  procedure CheckBox4_Click(sender: System.Object;
                            e: System.EventArgs);
  procedure Button1_Click(sender: System.Object;
                          e: System.EventArgs);
  procedure RadioButton1_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton2_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton3_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton4_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton5_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton6_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton7_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton8_Click(sender: System.Object;
                               e: System.EventArgs);
 {$ENDREGION}
 strict protected
  procedure Dispose(Disposing: Boolean); override;
 private
  { Private Declarations }
  sum:integer;      // Суммарная стоимость заказа.
  s1,s2,s3,s4:integer; // Стоимость выбранных опций по категориям.
 public
  constructor Create;
 end;

 [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm1))]

implementation
```

```
{$AUTOBOX ON}

{$REGION 'Windows Form Designer generated code'}

procedure TWinForm1.Dispose(Disposing: Boolean);
begin
 if Disposing then
 begin
  if Components <> nil then
   Components.Dispose();
 end;

 inherited Dispose(Disposing);
end;
constructor TWinForm1.Create;
begin
 inherited Create;
 InitializeComponent;
 sum:=0; // Начальное значение стоимости заказа.
end;

// Определение, выбран ли пункт 'Кофе'.
procedure TWinForm1.RadioButton8_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 if RadioButton8.Checked then s4:=20;
end;

// Определение, выбран ли пункт 'Чай'.
procedure TWinForm1.RadioButton7_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 if RadioButton7.Checked then s4:=10;
end;

// Определение, выбран ли пункт 'Торт'.
procedure TWinForm1.RadioButton6_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 if RadioButton6.Checked then s3:=45;
end;

// Определение, выбран ли пункт 'Пирожное'.
procedure TWinForm1.RadioButton5_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 if RadioButton5.Checked then s3:=30;
end;
```

```
// Определение, выбран ли пункт 'Пельмени'.
procedure TWinForm1.RadioButton4_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 if RadioButton4.Checked then s2:=35;
end;

// Определение, выбран ли пункт 'Плов'.
procedure TWinForm1.RadioButton3_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 if RadioButton3.Checked then s2:=55;
end;

// Определение, выбран ли пункт 'Гороховый суп'.
procedure TWinForm1.RadioButton2_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 if RadioButton2.Checked then s1:=40;
end;

// Определение, выбран ли пункт 'Борщ'.
procedure TWinForm1.RadioButton1_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 if RadioButton1.Checked then s1:=50;
end;

// Расчет суммарной стоимости заказа и вывод значения в Label2.
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
 sum:=s1+s2+s3+s4;
 Label2.Text:='Ваш заказ составил: '+sum.ToString+' рублей';
end;

// Нажатие на переключатель CheckBox4.
procedure TWinForm1.CheckBox4_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 GroupBox4.Enabled:=CheckBox4.Checked;
 if not GroupBox4.Enabled then
  begin
   RadioButton7.Checked:=True;
   s4:=0;
  end
  else s4:=10;
```

```
end;

// Нажатие на переключатель CheckBox3.
procedure TWinForm1.CheckBox3_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 GroupBox3.Enabled:=CheckBox3.Checked;
 if not GroupBox3.Enabled then
  begin
   RadioButton5.Checked:=True;
   s3:=0;
  end
  else s3:=30;
end;

// Нажатие на переключатель CheckBox2.
procedure TWinForm1.CheckBox2_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 GroupBox2.Enabled:=CheckBox2.Checked;
 if not GroupBox2.Enabled then
  begin
   RadioButton3.Checked:=True;
   s2:=0;
  end
  else s2:=55;
end;

// Нажатие на переключатель CheckBox1.
procedure TWinForm1.CheckBox1_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 GroupBox1.Enabled:=CheckBox1.Checked;
 if not GroupBox1.Enabled then
  begin
   RadioButton1.Checked:=True;
   s1:=0;
  end
  else s1:=50;
end;

end.
```

Процедура обработки события CheckedChanged делает доступным или недоступным соответствующую панель. С помощью переключателей CheckBox устанавливается, что будет заказано (первое блюдо, второе и т.д.), а с помощью

компонента RadioButton (описание компонента будет рассмотрено ниже) выбирается, что именно будет заказано. Вывод стоимости заказа в поле Label2 осуществляет процедура TWinForm1.Button1_Click компонента Button1.

Компонент RadioButton

Компонент RadioButton представляет собой группу кнопок (или переключателей) с поясняющим текстом, который обычно располагается справа. Состояние кнопки зависит от состояния других кнопок (компонентов RadioButton). В каждый момент времени в выбранном состоянии может находиться только одна из кнопок, находящихся на форме. Однако возможна ситуация, когда ни одна из кнопок не выбрана. В поле компонента, помимо текста, могут также присутствовать и картинки.

Несколько компонентов RadioButton можно объединить в группу, разместив их в поле компонента GroupBox. При этом состояние компонентов, принадлежащих одной группе, не зависит от состояния компонентов, принадлежащих другой группе.

В табл. 5.8 приведены основные свойства компонента RadioButton.

Таблица 5.8 ▾ Основные свойства компонента RadioButton

Свойство	Комментарий
Name	Имя компонента, используемое для доступа в программе к компоненту и его свойствам
Text	Текст, располагающийся справа от кнопки
Checked	Свойство определяет, в каком состоянии находится переключатель. Если переключатель выбран, то значение свойства равно True, если не выбран – False
Enabled	Свойство, определяющее, доступен ли переключатель (Enabled=True) или нет (Enabled=False)
Visible	Свойство позволяет отображать (Visible=True) и скрывать (Visible=False) панель
CheckAlign	Положение кнопки в поле компонента. Свойство может принимать следующие значения: MiddleLeft, MiddleCenter, MiddleRight, TopLeft, TopCenter, TopRight, BottomLeft, BottomCenter, BottomRight
TextAlign	Положение текста в поле отображения текста. Свойство может принимать следующие значения: MiddleLeft, MiddleCenter, MiddleRight, TopLeft, TopCenter, TopRight, BottomLeft, BottomCenter, BottomRight
Image	Картинка, отображаемая в поле компонента
ImageAlign	Положение картинки в поле компонента. Свойство может принимать следующие значения: MiddleLeft, MiddleCenter, MiddleRight, TopLeft, TopCenter, TopRight, BottomLeft, BottomCenter, BottomRight
BackgroundImage	Картинка, определяющая фон компонента

Таблица 5.8 ▼ Основные свойства компонента `RadioButton` (окончание)

Свойство	Комментарий
`ImageList`	Набор картинок, используемых для обозначения различных состояний кнопки. Свойство содержит ссылку (имя) на объект типа `ImageList`
`ImageIndex`	Номер (индекс) картинки из набора `ImageList`
`Appearance`	Свойство определяет вид переключателя. Переключатель может быть обычным (`Normal`) или выполненным в виде кнопки (`Button`)
`AutoCheck`	Свойство определяет, будет ли автоматически изменяться состояние переключателя в результате щелчка по его изображению. По умолчанию значение этого свойства установлено в `True`
`FlatStyle`	Задает стиль переключателя. Переключатель может быть стандартным (`Normal`), плоским (`Flat`) или «всплывающим» (`Popup`). Стиль отображения проявляется при наведении на переключатель указателя мыши

Состояние кнопки изменяется в результате щелчка по ее изображению (если значение свойства `AutoCheck` равно `True`). При этом возникает событие `CheckedChanged` и затем событие `Click`. Если значение свойства `AutoCheck` равно `False`, то в результате щелчка на переключателе возникает событие `Click`, а затем, если процедура обработки изменит состояние переключателя, возникает событие `CheckedChanged`.

Следующая программа, окно которой приведено на рис. 5.7, демонстрирует использование компонента `RadioButton`.

Текст программы приведен в листинге 5.5. Программа работает следующим образом. При запуске программы в переменные `country`, `hotel`, `food` и `ex` заносятся начальные значения. При нажатии на определенный компонент `RadioButton` значение соответствующей переменной изменяется на значения

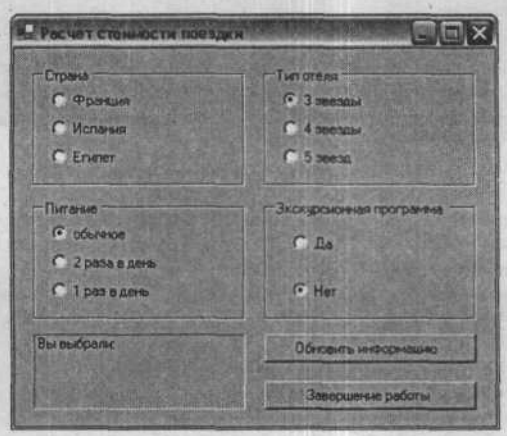

Рис. 5.7 ▼ Окно программы, демонстрирующей использование компонента `RadioButton`

свойства Text этого компонента. Процедура вывода выбранных опций закреплена за событием Click кнопки Button1.

Следует заметить, что после запуска программы в группе кнопок **Страна** ни один из элементов не выбран, поэтому при выводе дополнительно проводится проверка, была ли выбрана страна или нет.

Листинг 5.5 ▾ Программа выбора опций поездки

```
unit WinForm1;

interface

uses
 System.Drawing, System.Collections, System.ComponentModel,
 System.Windows.Forms, System.Data;

type
 TWinForm1 = class(System.Windows.Forms.Form)
 {$REGION 'Designer Managed Code'}
 strict private
  Components: System.ComponentModel.Container;
  GroupBox1: System.Windows.Forms.GroupBox;
  GroupBox2: System.Windows.Forms.GroupBox;
  GroupBox3: System.Windows.Forms.GroupBox;
  GroupBox4: System.Windows.Forms.GroupBox;
  RadioButton1: System.Windows.Forms.RadioButton;
  RadioButton2: System.Windows.Forms.RadioButton;
  RadioButton3: System.Windows.Forms.RadioButton;
  RadioButton4: System.Windows.Forms.RadioButton;
  RadioButton5: System.Windows.Forms.RadioButton;
  RadioButton6: System.Windows.Forms.RadioButton;
  RadioButton7: System.Windows.Forms.RadioButton;
  RadioButton8: System.Windows.Forms.RadioButton;
  RadioButton9: System.Windows.Forms.RadioButton;
  RadioButton10: System.Windows.Forms.RadioButton;
  RadioButton11: System.Windows.Forms.RadioButton;
  Label1: System.Windows.Forms.Label;
  Button1: System.Windows.Forms.Button;
  Button2: System.Windows.Forms.Button;
  procedure InitializeComponent;
  procedure Button1_Click(sender: System.Object;
                          e: System.EventArgs);
  procedure RadioButton1_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton2_Click(sender: System.Object;
                               e: System.EventArgs);
```

```
  procedure RadioButton3_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton4_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton5_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton6_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton7_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton8_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton9_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton10_Click(sender: System.Object;
                                e: System.EventArgs);
  procedure RadioButton11_Click(sender: System.Object;
                                e: System.EventArgs);
  procedure TWinForm1_Paint(sender: System.Object;
                            e: System.Windows.Forms.PaintEventArgs);
  procedure Button2_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(TWinForm1))]

implementation

var country:string;  // Переменная для хранения информации
                     // о выбранной стране.
  hotel:string;  // Переменная для хранения информации о выбранном
                 // отеле.
  food:string;   // Переменная для хранения информации о выбранном
                 // рационе.
  ex:string;     // Переменная для хранения информации об экскурс.
                 // программе.

{$AUTOBOX ON}

{$REGION 'Windows Form Designer generated code'}

procedure TWinForm1.Dispose(Disposing: Boolean);
```

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

constructor TWinForm1.Create;
begin
 inherited Create;
 InitializeComponent;
end;

// Нажатие на кнопку 'Завершение работы'.
procedure TWinForm1.Button2_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
 // Завершение работы программы.
 Close;
end;

// Выполнение начальных установок в момент появления
// окна на экране.
procedure TWinForm1.TWinForm1_Paint(sender: System.Object;
                        e: System.Windows.Forms.PaintEventArgs);
begin
 // Устанавливаем начальные значения переменных.
 country:='';
 hotel:=RadioButton4.Text;
 food:=RadioButton7.Text;
 ex:=RadioButton11.Text;
end;

// Нажатие на кнопку 'Нет' группы 'Экскурсионная программа'.
procedure TWinForm1.RadioButton11_Click(sender: System.Object;
                                        e: System.EventArgs);
begin
 ex:=RadioButton11.Text;
end;

// Нажатие на кнопку 'Да' группы 'Экскурсионная программа'.
procedure TWinForm1.RadioButton10_Click(sender: System.Object;
                                        e: System.EventArgs);
begin
 ex:=RadioButton10.Text;
end;
```

```
// Нажатие на кнопку '1 раз в день' группы 'Питание'.
procedure TWinForm1.RadioButton9_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 food:=RadioButton9.Text;
end;

// Нажатие на кнопку '2 раза в день' группы 'Питание'.
procedure TWinForm1.RadioButton8_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 food:=RadioButton8.Text;
end;

// Нажатие на кнопку 'обычное' группы 'Питание'.
procedure TWinForm1.RadioButton7_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 food:=RadioButton7.Text;
end;

// Нажатие на кнопку '5 звезд' группы 'Отель'.
procedure TWinForm1.RadioButton6_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 hotel:=RadioButton6.Text;
end;

// Нажатие на кнопку '4 звезды' группы 'Отель'.
procedure TWinForm1.RadioButton5_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 hotel:=RadioButton5.Text;
end;

// Нажатие на кнопку '3 звезды' группы 'Отель'.
procedure TWinForm1.RadioButton4_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 hotel:=RadioButton4.Text;
end;

// Нажатие на кнопку 'Египет' группы 'Страна'.
procedure TWinForm1.RadioButton3_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 country:=RadioButton3.Text;
end;
```

```
// Нажатие на кнопку 'Испания' группы 'Страна'.
procedure TWinForm1.RadioButton2_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 country:=RadioButton2.Text;
end;

// Нажатие на кнопку 'Франция' группы 'Страна'.
procedure TWinForm1.RadioButton1_Click(sender: System.Object;
                                       e: System.EventArgs);
begin
 country:=RadioButton1.Text;
end;

// Нажатие на кнопку 'Обновить информацию'.
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
 // Проверяем, была ли введена страна.
 if country='' then
  begin
   // Если страна не введена, то выводим соответствующее сообщение
   MessageBox.Show('Необходимо указать страну','Ошибка при вводе',
   MessageBoxButtons.OK,MessageBoxIcon.Information);
   // и выходим из процедуры отображения выбранных опций.
   Exit
  end;

 // Если страна введена, то отображаем выбранные опции в Label1.
 Label1.Text:='Вы выбрали: страна - '+country+
 ', отель - '+hotel+
 ', питание - '+food+
 ', экс. программа - '+ex;
end;

end.
```

Компонент GroupBox

Этот компонент мы уже упоминали ранее и даже использовали в предыдущих примерах, так что настало время познакомиться с ним поближе. Компонент GroupBox представляет собой контейнер для других компонентов. Обычно он используется для объединения компонентов в группы по функциональным признакам.

Основные свойства компонента приведены в табл. 5.9.

Таблица 5.9 ▼ Основные свойства компонента `GroupBox`

Свойство	Комментарий
`Name`	Имя компонента, используемое для доступа в программе к компоненту и его свойствам
`Text`	Текст, располагающийся в верхней части. Основное назначение свойства состоит в пояснении предназначения компонентов, находящихся в группе
`Font`	Шрифт панели и всех входящих в нее компонентов
`Enabled`	Свойство, определяющее, доступна ли панель (`Enabled=True`) или нет (`Enabled=False`)
`Visible`	Свойство позволяет отображать (`Visible=True`) и скрывать (`Visible=False`) панель

Компонент ComboBox

Компонент `ComboBox` представляет собой сочетание поля редактирования и списка, что позволяет вводить данные путем набора с клавиатуры или выбором из списка. Свойства компонента приведены в табл. 5.10.

Таблица 5.10 ▼ Основные свойства компонента `ComboBox`

Свойство	Комментарий
`Name`	Имя компонента, используемое для доступа в программе к компоненту и его свойствам
`Text`	Текст, располагающийся в поле для ввода (если компонент находится в режиме `DropDown` или `Simple`)
`Font`	Шрифт текста
`Enabled`	Свойство, определяющее, доступен ли компонент для ввода/выбора (`Enabled=True`) или нет (`Enabled=False`)
`Visible`	Свойство, позволяющее отображать (`Visible=True`) и скрывать (`Visible=False`) компонент
`DropDownStyle`	Свойство, определяющее вид компонента: `DropDown` – поле ввода и раскрывающийся список, `Simple` – поле ввода со списком, `DropDownList` – раскрывающийся список
`MaxDropDownItems`	Максимальное количество показываемых элементов в выпадающем списке
`DropDownWidth`	Ширина области списка
`Items`	Элементы списка – коллекция строк
`Items.SelectedIndex`	Номер элемента, который в данный момент выбран в списке. Если выбранного элемента нет, то значение свойства равно –1
`Items.Count`	Общее количество элементов списка
`Sorted`	Признак необходимости сортировки элементов коллекции после добавления очередного элемента
`Location.X`, `Location.Y`	Свойство, определяющее положение компонента на поверхности формы
`Size.Width` `Size.Height`	Размер компонента с учетом (для типа `Simple`) или без учета (для типов `DropDown` и `DropDownList`) размера области списка или области ввода

Список, отображаемый в поле компонента, можно формировать во время создания формы или во время работы программы.

Чтобы сформировать список во время работы программы, надо вызвать метод `Add` у свойства `Items` (которое является в свою очередь объектом). Например, следующий фрагмент кода формирует упорядоченный по алфавиту список:

```
// Метод очистки элементов списка.
ComboBox1.Items.Clear;
// Установка признака необходимости сортировки.
ComboBox1.Sorted:=True;
// Заполнение элементов списка.
ComboBox1.Text:='Англия';
ComboBox1.Items.Add('Англия');
ComboBox1.Items.Add('Франция');
ComboBox1.Items.Add('Германия');
ComboBox1.Items.Add('Италия');
ComboBox1.Items.Add('Греция');
ComboBox1.Items.Add('Испания');
```

А на экране сформированный подобным образом список выглядел бы, как показано на рис. 5.8.

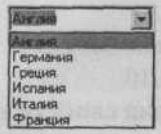

Рис. 5.8 ▾ Результат применения метода `Add` к свойству `Items`

Аналогичного результата можно добиться, если использовать для ввода элементов списка редактор свойств **String Collection Editor**. С помощью этого редактора список можно сформировать во время создания формы. Для этого необходимо щелкнуть по кнопке с тремя точками в строке свойства **Items** (Элементы) и в появившемся окне **String Collection Editor** (рис. 5.9) ввести элементы списка.

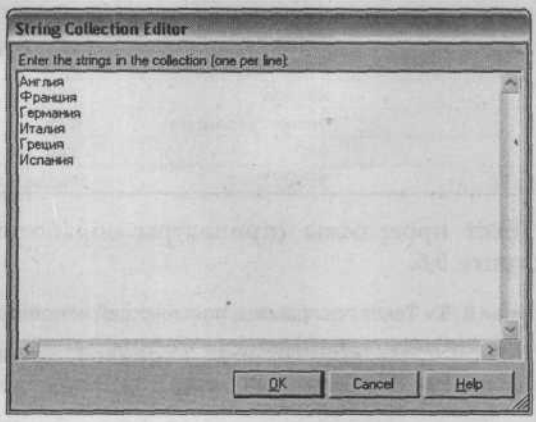

Рис. 5.9 ▾ Формирование списка компонента `ComboBox` во время создания формы

Следующая программа демонстрирует использование

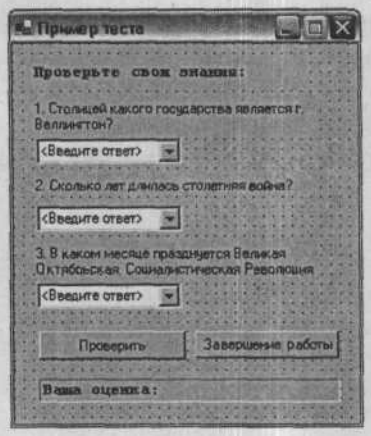

Рис. 5.10 ▼ Форма приложения, демонстрирующего использование компонента ComboBox

компонента ComboBox для ввода данных. Форма приложения приведена на рисунке 5.10.

Значения свойств компонентов ComboBox приведены в табл. 5.11.

Таблица 5.11 ▼ Свойства компонентов ComboBox

Свойство	ComboBox1	ComboBox2	ComboBox3
Text	<Введите ответ>	<Введите ответ>	<Введите ответ>
Items	Австралия	98	октябрь
	Великобритания	100	ноябрь
	Канада	116	декабрь
	Новая Зеландия	135	январь
	ЮАР	140	февраль
Sorted	True	False	False

Текст программы (процедуры обработки события Click) приведен в листинге 5.6.

Листинг 5.6 ▼ Текст программы, поясняющей использование компонента ComboBox

```
// Процедура выставления итоговой оценки за тест.
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var
 mark:integer; // Оценка за тест.
begin
```

```
// Ввод начального значения оценки
// (каждый неправильный ответ уменьшает ее на 1).
mark:=5;
// Проверка ввода всех ответов в ComboBox1-ComboBox3.
if (ComboBox1.SelectedIndex=-1) or
   (ComboBox2.SelectedIndex=-1) or
   (ComboBox3.SelectedIndex=-1)
  then
   begin
    MessageBox.Show('Необходимо указать все ответы',
            'Ошибка при вводе данных',
            MessageBoxButtons.OK,
            MessageBoxIcon.Information);
    Exit;
   end;
// Формирование оценки.
if ComboBox1.SelectedIndex<>3
  then mark:=mark-1;
if ComboBox2.SelectedIndex<>2
  then mark:=mark-1;
if ComboBox3.SelectedIndex<>1
  then mark:=mark-1;
Label5.Text:='Ваша оценка: '+mark.ToString;
end;
```

Программа работает следующим образом. Изначально в переменную, которая будет содержать оценку, заносится значение 5. Затем осуществлется проверка – были ли введены данные во все поля для ввода. Для этого используется свойство `ItemIndex` компонента `ComboBox`, которое содержит –1, если элемент списка не выбран. В случае отсутствия хотя бы одного ответа выводится информационное сообщение и производится досрочное завершение процедуры. Если все ответы введены, то осуществляется последовательное сравнение введенных ответов с правильными. Каждый неправильный ответ уменьшает оценку (значение переменной `mark`) на единицу.

Компонент ListBox

Данный компонент представляет собой список, в котором пользователь может выбирать нужный элемент. Основные свойства компонента приведены в табл. 5.12.

Таблица 5.12 ▼ Основные свойства компонента `ListBox`

Свойство	Комментарий
`Name`	Имя компонента, используемое для доступа в программе к компоненту и его свойствам
`Items`	Элементы списка – коллекция строк
`Items.SelectedIndex`	Номер элемента, который в данный момент выбран в списке. Если выбранного элемента нет, то значение свойства равно -1
`Items.Count`	Общее количество элементов списка
`Sorted`	Признак необходимости (`Sorted=True`) сортировки элементов коллекции после добавления очередного элемента
`SelectionMode`	Свойство определяет режим выбора элементов списка. Может принимать следующие значения: `One` – только один элемент, `MultiSimple` – возможен выбор нескольких элементов, `MultiExtended` – расширенный вариант выбора нескольких элементов списка. В последнем случае несколько элементов можно выбирать нажатием клавиши **Ctrl**, а также выбирать диапазон с помощью нажатой клавиши **Shift**
`ScrollAlwaysVisible`	Признак необходимости всегда отображать вертикальную полосу прокрутки. Если свойство равно `False`, то полоса прокрутки отображаться не будет, в результате чего не будут отображаться некоторые элементы списка
`MultiColumn`	Признак необходимости отображать список в несколько колонок. Количество колонок зависит от количества элементов и размера компонента
`Font`	Шрифт, используемый для вывода элементов списка
`Location.X`, `Location.Y`	Свойство, определяющее положение компонента на поверхности формы
`Size.Width` `Size.Height`	Размер компонента с учетом (для типа `Simple`) или без учета (для типов `DropDown` и `DropDownList`) размера области списка или области ввода

Список, отображаемый в поле редактирования, можно сформировать как при создании формы, так и при выполнении программы. В первом случае необходимо щелкнуть по кнопке с тремя точками в поле свойства **Items** и в окне редактора свойств **String Collection Editor** (рис. 5.11) ввести элементы списка. Во втором случае необходимо вызвать уже знакомый нам метод `Add` свойства `Items`.

Далее приведена программа, демонстрирующая пример использования этого компонента. Программа просматривает файлы изображений, находящихся в выбранном каталоге.

Форма программы приведена на рис. 5.12, а текст – в листинге 5.7.

Программа работает следующим образом. Элементы списка `ListBox` заполняются во время выполнения программы. Заполняет список компонента `ListBox` процедура обработки события `Load` формы. Для доступа к выбранной папке используется объект `DirectoryInfo`. Список файлов представлен в виде массива `FileInfo`. Формирует массив метод `GetFiles`. Для отображения файлов

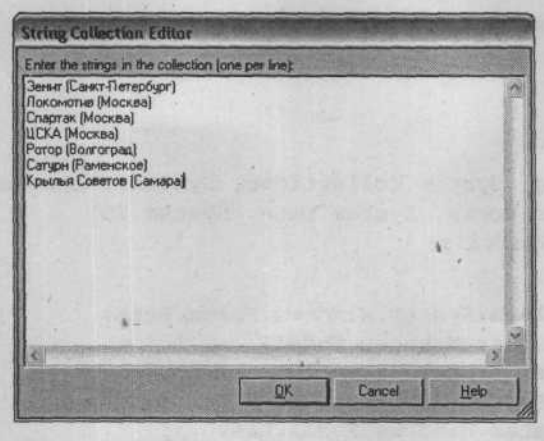

Рис. 5.11 ▾ Ввод элементов списка с помощью окна редактора свойств **String Collection Editor**

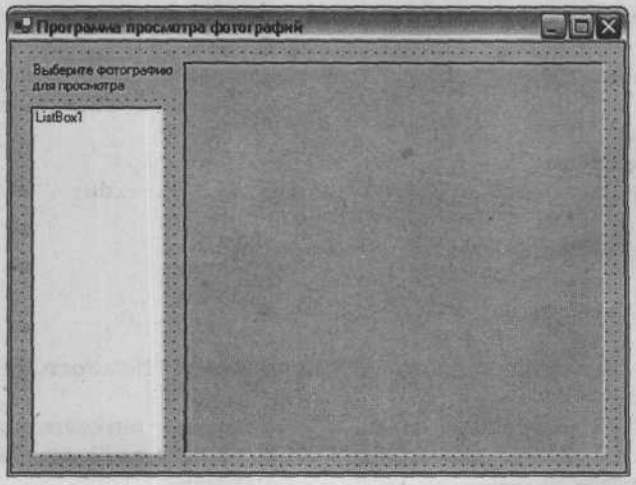

Рис. 5.12 ▾ Форма программы просмотра изображений

изображений используется компонент `PictureBox`. Отображение выбранного файла в списке осуществляет процедура обработки события `SelectedIndexChenged`, которое происходит в результате щелчка по элементу списка или перемещения указателя текущего элемента списка при помощи клавиш клавиатуры.

Листинг 5.7 ▼ Текст программы для просмотра изображений

```
unit WinForm1;

interface

uses
 System.Drawing, System.Collections, System.ComponentModel,
 System.Windows.Forms, System.Data, System.IO,
  Borland.Vcl.SysUtils;

type
 TWinForm1 = class(System.Windows.Forms.Form)
 {$REGION 'Designer Managed Code'}
 strict private
  Components: System.ComponentModel.Container;
  ListBox1: System.Windows.Forms.ListBox;
  Label1: System.Windows.Forms.Label;
  PictureBox1: System.Windows.Forms.PictureBox;
  procedure InitializeComponent;
  procedure TWinForm1_Load(sender: System.Object;
                           e: System.EventArgs);
  procedure ListBox1_SelectedIndexChanged(sender: System.Object;
                                          e: System.EventArgs);
 {$ENDREGION}
 strict protected
  procedure Dispose(Disposing: Boolean); override;
 private
  { Private Declarations }
 public
  constructor Create;
 end;
 [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm1))]
implementation
var current_dir:DirectoryInfo;    // Содержит информацию о каталоге.
  dir_files:array of FileInfo;    // Содержит информацию о файлах
                                  // каталога.
{$AUTOBOX ON}
procedure TWinForm1.Dispose(Disposing: Boolean);
begin
 if Disposing then
 begin
  if Components <> nil then
   Components.Dispose();
 end;
```

```
 inherited Dispose(Disposing);
end;
constructor TWinForm1.Create;
begin
 inherited Create;
 InitializeComponent;
end;
procedure TWinForm1.TWinForm1_Load(sender: System.Object;
                                   e: System.EventArgs);
var i:integer;  // Счетчик цикла.
  num:integer;  // Переменная для хранения количества файлов
                // в каталоге.
  s:string; // Переменная для хранения имени текущего каталога.
begin
  // Узнаем текущий каталог программы.
  s:=GetCurrentDir();
  // Создаем объект типа DirectoryInfo.
  try
   current_dir:=DirectoryInfo.Create(s+'\Photos\');
  except
   On e:Exception do
    begin
     MessageBox.Show(e.Message);
    end;
  end;
  // Читаем файлы *.jpg из подкаталога photos.
  dir_files:=current_dir.GetFiles('*.jpg');
  // Запоминаем количество найденных файлов.
  num:=length(dir_files);
  // Устанавливаем признак необходимости сортировки в True.
  ListBox1.Sorted:=True;
  // Добавляем найденные файлы в ListBox.
  for i:=0 to num-1 do
   ListBox1.Items.Add(dir_files[i].Name);
  // Ставим указатель на первый элемент списка.
  ListBox1.SelectedIndex:=0;
end;

// Процедура отображения текущего элемента списка (файла)
// в компоненте PictureBox.
procedure TWinForm1.ListBox1_SelectedIndexChanged(sender:
                                  System.Object; e: System.EventArgs);
var str:string;
begin
```

```
  // Формируем полное имя файла, который будем отображать.
  str:=current_dir.FullName+ListBox1.SelectedItem.ToString();
  // Заносим это имя в свойство Image компонента PictureBox.
  PictureBox1.Image:=System.Drawing.Bitmap.FromFile(str);
end;

end.
```

Результат работы программы приведен на рис. 5.13.

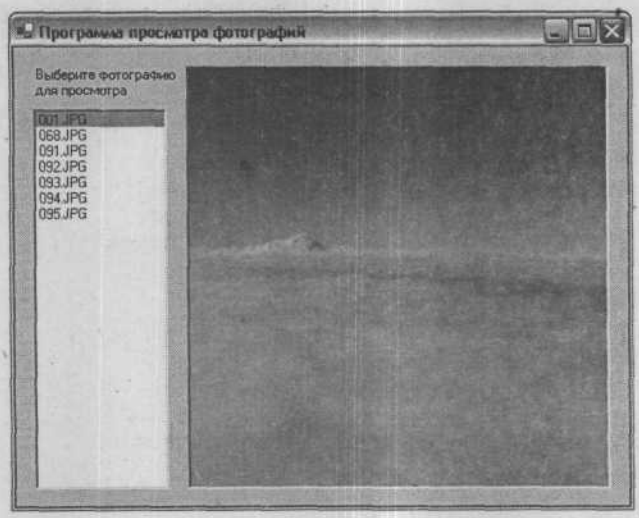

Рис. 5.13 ▼ Результат работы программы просмотра изображений

Компонент CheckedListBox

Этот компонент также является списком, однако перед каждым его элементом находится переключатель CheckBox. Свойства компонента приведены в табл. 5.13.

Таблица 5.13 ▼ Основные свойства компонента CheckedListBox

Свойство	Комментарий
Name	Имя компонента, используемое для доступа в программе к компоненту и его свойствам
Items	Элементы списка – коллекция строк
Items.Count	Общее количество элементов списка

Таблица 5.13 ▾ Основные свойства компонента `CheckedListBox` (окончание)

Свойство	Комментарий
`CheckOnClick`	Способ пометки элементов списка. Если свойство установлено в `True`, то при нажатии на элемент списка производится его выбор и изменение флажка переключателя. Если же свойство установлено в `False`, то при нажатии происходит выбор элемента списка, а изменение состояния переключателя – только при повторном нажатии на элемент списка
`CheckedItems`	Коллекция, содержащая выбранные элементы списка
`CheckedItems.Count`	Количество выбранных элементов списка
`CheckedIndices`	Коллекция, содержащая номера выбранных элементов списка
`Sorted`	Признак необходимости (`Sorted=True`) сортировки элементов коллекции после добавления очередного элемента
`SelectionMode`	Свойство определяет режим выбора элементов списка. Может принимать следующие значения: `One` – только один элемент, `MultiSimple` – возможен выбор нескольких элементов, `MultiExtended` – расширенный вариант выбора нескольких элементов списка. В последнем случае несколько элементов можно выбирать нажатием клавиши **Ctrl**, а также выбирать диапазон с помощью нажатой клавиши **Shift**
`ScrollAlwaysVisible`	Признак необходимости всегда отображать вертикальную полосу прокрутки. Если свойство равно `False`, то полоса прокрутки отображаться не будет, что может привести к тому, что не будут отображаться некоторые элементы списка
`MultiColumn`	Признак необходимости отображать список в несколько колонок. Количество колонок зависит от количества элементов и размера компонента
`Location.X`, `Location.Y`	Свойство, определяющее положение компонента на поверхности формы
`Size.Width` `Size.Height`	Размер компонента с учетом (для типа `Simple`) или без учета (для типов `DropDown` и `DropDownList`) размера области списка или области ввода
`Font`	Шрифт, используемый для отображения содержимого поля редактирования и элементов списка

Формируется список `CheckedListBox` также двумя способами – либо с использованием редактора свойств **String Collection Editor**, либо с использованием метода `Add` свойства `Items`.

Компонент PictureBox

Компонент `PictureBox` обеспечивает отображение иллюстрации (файла рисунка). Свойства компонента приведены в табл. 5.14.

Таблица 5.14 ▾ Основные свойства компонента `PictureBox`

Свойство	Комментарий
`Name`	Имя компонента, используемое для доступа в программе к компоненту и его свойствам
`Image`	Иллюстрация, отображаемая в поле компонента

Таблица 5.14 ▼ Основные свойства компонента `PictureBox` (окончание)

Свойство	Комментарий
`Image.PhysicalDimension`	Свойство содержит информацию о размере иллюстрации, загруженной в поле компонента
`SizeMode`	Способ отображения картинки. Существует четыре режима отображения – так называемые способы масштабирования. `Normal` – обычный режим, без масштабирования, `StretchImage` – масштабирование картинки до размера области отображения компонента `PictureBox` (если размер картинки не пропорционален размеру иллюстрации, то она будет искажена), `AutoSize` – размер компонента автоматически изменяется до размеров картинки, `CenterImage` – центрирование картинки в поле отображения компонента, если размер иллюстрации меньше размера области отображения
`BorderStyle`	Вид границы компонента: `None` (граница отсутствует), `FixedSingle` (обычная), `Fixed3D` («объемная» граница)
`Location.X`, `Location.Y`	Свойство, определяющее положение компонента на поверхности формы
`Size.Width` `Size.Height`	Размер компонента
`Visible`	Признак необходимости отображения компонента (и, соответственно, изображения)
`Graphics`	Поверхность, на которую можно выводить графику

Чтобы задать картинку во время дизайна формы, необходимо в строке свойства **Image** щелкнуть на кнопке с тремя точками и в появившемся окне выбрать файл изображения. Добавленное таким образом в форму изображение будет обработано Delphi и помещено в файл ресурсов проекта. Соответственно, сам файл изображения нам больше не понадобится. Если же потребуется загружать картинку во время выполнения программы, то следует воспользоваться методом `FromFile` (дословно – «Из файла»). В качестве параметра нужно указать полное имя загружаемого в `PictureBox` файла. Например, инструкция

```
PictureBox1.Image:=System.Drawing.Bitmap.FromFile('d:\example.bmp')
```

загрузит и отобразит в компоненте `PictureBox` файл d:\example.bmp. Компонент может отображать не только файлы формата bmp. Данным методом можно также загружать файлы jpg, gif, png.

Помните, что при загрузке изображения компонент `PictureBox` не обеспечивает пропорционального масштабирования (без искажения).

В качестве примера можно привести программу, иллюстрирующую особенности использования компонента `PictureBox`. Форма программы приведена на рис. 5.14.

При выборе соответствующего переключателя устанавливаются режимы масштабирования. Всего таких режимов четыре:

- `Normal` – масштабирование не производится;
- `CenterImage` – изображение центрируется;

Рис. 5.14 ▾ Форма программы, демонстрирующей использование компонента `PictureBox`

- `StretchImage` – изображение «подгоняется» под размер компонента;
- `AutoSize` – автоматическое изменение размеров области отображения в зависимости от размера картинки.

Текст программы приведен в листинге 5.8.

Листинг 5.8 ▾ Текст программы, демонстрирующей использование компонента `PictureBox`

```
unit WinForm1;

interface

uses
 System.Drawing, System.Collections, System.ComponentModel,
 System.Windows.Forms, System.Data, System.Resources;

type
 TWinForm1 = class(System.Windows.Forms.Form)
 {$REGION 'Designer Managed Code'}
 strict private
  Components: System.ComponentModel.Container;
  PictureBox1: System.Windows.Forms.PictureBox;
  GroupBox1: System.Windows.Forms.GroupBox;
  RadioButton1: System.Windows.Forms.RadioButton;
  RadioButton2: System.Windows.Forms.RadioButton;
  RadioButton3: System.Windows.Forms.RadioButton;
  RadioButton4: System.Windows.Forms.RadioButton;
```

```
  RadioButton5: System.Windows.Forms.RadioButton;
  procedure InitializeComponent;
  procedure RadioButton2_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton1_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton3_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton4_Click(sender: System.Object;
                               e: System.EventArgs);
  procedure RadioButton5_CheckedChanged(sender: System.Object;
                                        e: System.EventArgs);
 {$ENDREGION}
 strict protected
  procedure Dispose(Disposing: Boolean); override;
 private
  P_Size : Size;    // Размер компонента PictureBox,
                    // задаваемый во время создания формы.
 public
  constructor Create;
 end;

 [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm1))]

implementation

{$AUTOBOX ON}

procedure TWinForm1.Dispose(Disposing: Boolean);
begin
 if Disposing then
 begin
  if Components <> nil then
   Components.Dispose();
 end;
 inherited Dispose(Disposing);
end;

constructor TWinForm1.Create;
begin
 inherited Create;
 InitializeComponent;
 // Получаем фактические размеры компонента PictureBox.
 P_Size.Width:=PictureBox1.Width;
 P_Size.Height:=PictureBox1.Height;
end;

// Режим отображения - Normal.
```

```
procedure TWinForm1.RadioButton1_Click(sender: System.Object;
    e: System.EventArgs);
begin
  PictureBox1.SizeMode:=PictureBoxSizeMode.Normal;
  PictureBox1.Size:=P_Size;
end;

// Режим отображения - CenterImage.
procedure TWinForm1.RadioButton2_Click(sender: System.Object;
    e: System.EventArgs);
begin
  PictureBox1.SizeMode:=PictureBoxSizeMode.CenterImage;
  PictureBox1.Size:=P_Size;
end;

// Режим отображения - StretchImage.
procedure TWinForm1.RadioButton3_Click(sender: System.Object;
    e: System.EventArgs);
begin
  PictureBox1.SizeMode:=PictureBoxSizeMode.StretchImage;
  PictureBox1.Size:=P_Size;
end;

// Режим отображения - AutoSize.
procedure TWinForm1.RadioButton4_Click(sender: System.Object;
    e: System.EventArgs);
begin
  PictureBox1.SizeMode:=PictureBoxSizeMode.AutoSize;
end;

// Пропорциональное отображение картинки.
procedure TWinForm1.RadioButton5_CheckedChanged(sender:
System.Object; e: System.EventArgs);
var
 dx,dy,scale: real;
begin
  // Режим отображения - без масштабирования.
  PictureBox1.SizeMode:=PictureBoxSizeMode.Normal;
  PictureBox1.Size:=P_Size;
  // Если ширина компонента PictureBox меньше ширины иллюстрации,
  // то вычислить коэффициент масштабирования по длине
  if PictureBox1.Width>PictureBox1.Image.PhysicalDimension.Width
   then dx:=1
   else dx:=PictureBox1.Width
            /PictureBox1.Image.PhysicalDimension.Width;
  // и высоте.
```

```
  if PictureBox1.Height > PictureBox1.Image.PhysicalDimension.Height
   then dy:=1
   else dy:=PictureBox1.Height
            /PictureBox1.Image.PhysicalDimension.Height;
  // Определяем итоговый коэффициент масштабирования
  // (чтобы не было искажения картинки,
  // коэффициенты масштабирования должны быть одинаковыми
   // по обеим осям).
  if dx<dy
   then
    scale:=dx
   else
    scale:=dy;
  if scale=1
   then
    // Если масштабировать не надо, режим масштибирования -
     // Normal.
    PictureBox1.SizeMode := PictureBoxSizeMode.Normal
   else
    begin
     // В противном случае
     // выполняем масштабирование.
    PictureBox1.SizeMode:=PictureBoxSizeMode.StretchImage;
     PictureBox1.Width:=Round(PictureBox1.Image.PhysicalDimension.Width*scale);
     PictureBox1.Height:=Round(PictureBox1.Image.PhysicalDimension.Height*scale);
    end;
end;

end.
```

Компонент NumericUpDown

Этот компонент предназначен для ввода числовых данных. Мы уже вводили различные данные, казалось бы, зачем использовать еще один компонент? Компонент предназначен для ввода *только* числовых данных, причем вводить данные можно как с клавиатуры, так и с использованием стрелок **Увеличить/Уменьшить**. При вводе дробных чисел имеется возможность округления. Для этого необходимо использовать свойство `DecimalPlaces`, в котором следует указать количество цифр, до которого будет произведено округление. Если же это свойство равно нулю, то будет произведено округление до целого значения.

Основные свойства компонента приведены в табл. 5.15.

Таблица 5.15 ▾ Основные свойства компонента `NumericUpDown`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Value`	Значение поля для ввода компонента. В отличие от поля для ввода `TextBox` имеет тип данных `decimal`, а не `string`
`Minimum`	Минимальное значение, которое можно ввести в компонент
`Maximum`	Максимальное значение, которое можно ввести в компонент
`Increment`	Величина, на которую увеличивается (или уменьшается) значение свойства `Value` при нажатии на соответствующую стрелку
`DecimalPlaces`	Свойство определяет количество цифр в дробной части числа
`Thousandseparator`	Признак необходимости отделять пробелом тысячные разряды числа
`UpDownAlign`	Свойство определяет, с какой стороны будут отображаться стрелки **Увеличить**/**Уменьшить**: `Right` – справа, `Left` – слева

Компонент StatusBar

Компонент `StatusBar` представляет собой область (панель) для вывода служебной информации. Обычно такая панель располагается в нижней части окна программы и может разбиваться на несколько частей.

Свойства компонента `StatusBar` приведены в табл. 5.16.

Таблица 5.16 ▾ Основные свойства компонента `StatusBar`

Свойство	Комментарий
`Name`	Имя компонента. Это имя используется в программе для доступа к компоненту и его свойствам
`Text`	Текст, который содержится в поле компонента
`Font`	Шрифт, который используется для отображения текста
`Color`	Цвет фона поля компонента
`Left`	Расстояние от левой границы формы до левой границы компонента
`Top`	Расстояние от верхней границы формы до верхней границы компонента
`Width`	Ширина поля компонента
`Height`	Высота поля компонента
`BorderStyle`	Вид рамки компонента. По умолчанию задается обычная (`bsSingle`) рамка. Свойство также может принимать значение `None` (рамка отсутствует)

Для того чтобы разделить панель на несколько составных частей, необходимо в строке свойства **Panels** (Панели) нажать кнопку с тремя точками и в появившемся окне редактора свойств **StatusBar Collection Editor** щелкнуть на кнопке **Add** столько раз, сколько планируется использовать панелей (рис. 5.15). В этом же окне можно выполнить настройку каждой из

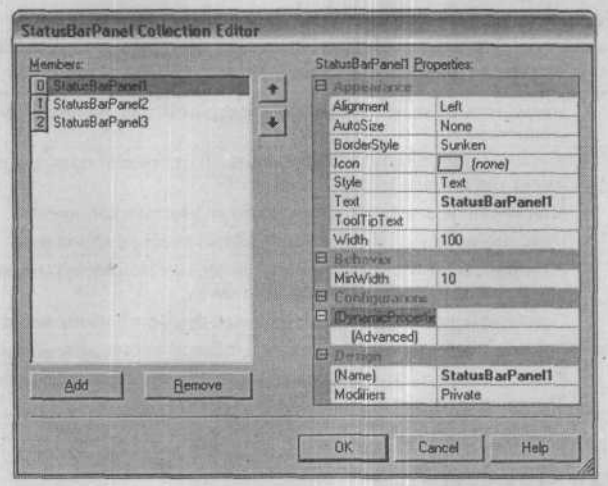

Рис. 5.15 ▼ Использование окна **StatusBar Collection Editor**

панелей. Свойства выбранной панели отображены в правой части окна **StatusBar Collection Editor**.

Основные свойства панелей объекта `StatusBar` приведены в табл. 5.17.

Таблица 5.17 ▼ Основные свойства панелей объекта `StatusBar`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Text`	Текст, отображаемый в панели
`Icon`	Картинка, отображаемая в панели
`Width`	Длина панели
`AutoSize`	Признак необходимости изменять размер панели. Если свойство имеет значение `Contents`, то ширина панели определяется ее содержанием (длиной текста). Если значение равно `None`, то панель имеет фиксированную ширину, определяемую свойством `Width`. Если свойство имеет значение `String`, то ширина панели выбирается такой, чтобы находящаяся справа панель была прижата к правому краю окна. Если справа панели нет, то ширина устанавливается такой, чтобы правая граница панели была прижата к правой границе окна
`MinWidth`	Минимальная длина панели (свойство имеет силу, если значение `AutoSize` не равно `None`)
`BorderStyle`	Вид границы панели. Панель может быть приподнята (`Raised`), утоплена (`Sunken`) или не иметь границы вообще (`None`)
`ToolTipText`	Свойство, определяющее текст всплывающей подсказки, появляющейся в случае наведения и задержки над панелью указателя мыши

Следующая программа, окно которой приведено на рис. 5.16, демонстрирует вариант применения этого компонента.

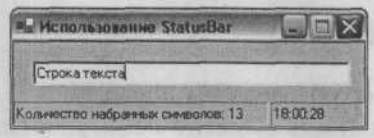

Рис. 5.16 ▾ Окно программы, поясняющей использование компонента `StatusBar`

Текст программы приведен ниже в листинге 5.9. Из особенностей работы программы можно отметить следующие. При появлении окна программы на экране выполняются начальные установки (см. обработку события `TWinForm1_Paint`) – в панели заносится текст, настраивается таймер (интервал возникновения события `Tick`). В конце процедуры таймер запускается и генерирует последовательность событий `Tick`. При возникновении каждого такого события формируется и отображается в правой части окна текущее время. Содержимое левой части панели (количество введенных символов) обновляется только при изменении поля для ввода `TextBox1` (см. процедуру обработки события `TextBox1_TextChanged`).

Листинг 5.9 ▾ Использование компонента `StatusBar` (основные процедуры)

```
procedure TWinForm1.TWinForm1_Paint(sender: System.Object;
                          e: System.Windows.Forms.PaintEventArgs);
begin
 // Начальные установки панелей - формирование текста.
 StatusBar1.Panels[0].Text:='Количество набранных символов: 0';
 StatusBar1.Panels[1].Text:=DateTime.Now.ToLongTimeString;
 // Установка интервала (1 секунда) и запуск таймера.
 Timer1.Interval:=1000;
 Timer1.Enabled:=True;
end;

// Обработка события таймера Tick - обновление значения времени.
procedure TWinForm1.Timer1_Tick(sender: System.Object;
                                e: System.EventArgs);
begin
 StatusBar1.Panels[1].Text:=DateTime.Now.ToLongTimeString;
end;

// Обновление содержимого левой панели при наборе текста.
procedure TWinForm1.TextBox1_TextChanged(sender: System.Object;
                                         e: System.EventArgs);
begin
 StatusBar1.Panels[0].Text:='Количество набранных символов: '+
                            TextBox1.TextLength.ToString;
end;
```

Компонент Timer

В примере из предыдущего раздела мы использовали компонент Timer, генерирующий последовательность событий Tick. Данный компонент невидим во время выполнения программы, то есть является *невизуальным*. Свойства компонента Timer приведены в табл. 5.18.

Таблица 5.18 ▼ Основные свойства компонента Timer

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Enabled	Свойство, определяющее, включен таймер или нет
Interval	Свойство определяет промежуток времени, через который происходит генерация события Tick. Задается в миллисекундах

Следующая программа (рис. 5.17), которая является простейшим секундомером, поясняет использование компонента Timer.

Сначала свойство Enabled компонента Timer установлено в False, поэтому таймер не генерирует никаких сообщений. Процедура обработки события Click на кнопке Button1 присваивает свойству Enabled значение True, тем самым запуская таймер. Процедура обработки события Tick отсчитывает интервалы с момента нажатия на кнопку **Пуск**. Нажатие на кнопку **Стоп** останавливает секундомер – опять же изменением свойства Enabled таймера Timer1. Текст программы приведен в листинге 5.10.

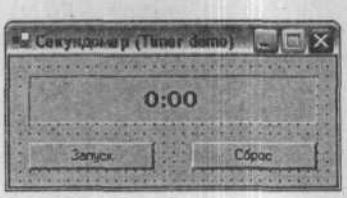

Рис. 5.17 ▼ Форма программы и использованием компонента Timer

Листинг 5.10 ▼ Текст программы (основные процедуры), поясняющей использование компонента Timer

```
// Процедура обработки события Click для левой кнопки.
// Кнопка имеет две функции - запуск и останов секундомера.
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
```

```
 if Timer1.Enabled
  then
    // Если секундомер работает.
    begin
     Timer1.Enabled := False; // Останавливаем таймер.
     Button1.Text := 'Пуск';  // Меняем название кнопки со 'Стоп'
                              // на 'Пуск'.
     Button2.Enabled := True; // Кнопка 'Сброс' теперь доступна.
    end
   else
    // Если секундомер не работает.
    begin
     Timer1.Enabled := True;  // Запускаем таймер.
     Button1.Text := 'Стоп';  // Меняем название кнопки с 'Пуск'
                              // на 'Стоп'.
     Button2.Enabled := False;// Кнопка 'Сброс' теперь недоступна.
    end;
end;

// Обработка события компонента Timer
// (обновление показаний секундомера).
procedure TWinForm1.Timer1_Tick(sender: System.Object;
                                e: System.EventArgs);
var
 str : string;
begin
  if sec = 59 then    // Если кол-во секунд равно 59,
  begin               // увеличиваем минуты на 1,
   inc(min);          // секунды обнуляем.
   sec := 0;
  end          // В противном случае просто
  else         // наращиваем секунды.
   inc(sec);
  // Формируем строку
  // в формате m:ss.
  str := str + Convert.ToString(sec);
  if str.Length = 1 then str := '0' + str;
  // Создаем эффект мигания двоеточия
  // (отображаем двоеточие только на четных секундах).
  if sec mod 2 = 0
   then str := ':' + str
   else str := ' ' + str;
  str := Convert.ToString(min) + str;
  Label1.Text := str;
```

```
end;

// Обработка нажатия на кнопку 'Сброс'.
procedure TWinForm1.Button2_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
 sec := 0;
 min := 0;
 Label1.Text := '0:00';
end;
```

Компонент ToolBar

Этот компонент представляет собой панель инструментов, на которой можно размещать *командные кнопки*. Свойства компонента `ToolBar` приведены в табл. 5.19.

Таблица 5.19 ▼ Основные свойства компонента `ToolBar`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Buttons`	Свойство задает кнопки панели инструментов (коллекцию объектов типа `ToolBarButton`)
`ButtonSize`	Размер находящихся на панели кнопок
`Appearance`	Свойство задает тип кнопки. Может принимать значения `Normal` (обычные кнопки), `Flat` (плоские кнопки)
`BorderStyle`	Определяет вид границы (рамки) панели инструментов. Граница панели инструментов может быть тонкой (`FixedSingle`), «объемной» (`Fixed3D`) или может отсутствовать (`None`)
`ImageList`	Свойство определяет компонент `ImageList`, который является источником картинок для кнопок панели
`TextAlign`	Свойство, определяющее положение поясняющего текста к кнопкам. Возможно два варианта положения текста: справа от картинки (`Right`), снизу картинки (`Under-neath`)
`Visible`	Признак необходимости отображать или скрывать панель инструментов. Если свойство установлено в `False`, то панель отображаться не будет
`Enabled`	Признак доступности кнопок панели инструментов. Если свойство установлено в `False`, то кнопки на панели будут недоступны
`Dock`	Свойство определяет границу окна, к которой будет прикреплена панель. Панель может прикрепляться к верхней (`Top`), нижней (`Bottom`), левой (`Left`) или правой (`Right`) границе

В качестве рекомендаций по использованию этого компонента можно отметить следующие. Перед использованием компонента лучше сначала добавить к форме компонент `ImageList` (см. выше) и произвести его настройку. После того как картинки к кнопкам будут определены, можно настраивать и саму панель с кнопками. Для этого нужно в строке свойства **Buttons** (Кнопки) щелкнуть по кнопке с тремя точками. В появившемся окне редактора свойств

ToolBarButton Collection Editor необходимо щелкнуть по кнопке **Add** для добавления кнопки на панель. После того как нужное количество кнопок добавлено, можно производить настройку каждой из них. Свойства кнопки указываются в правой части упомянутого выше окна (рис. 5.18).

В качестве примера использования этого компонента можно привести программу, являющуюся простейшим редактором текстовых файлов. На рис. 5.19 показана форма приложения, а в листинге 5.11 приведен текст программы.

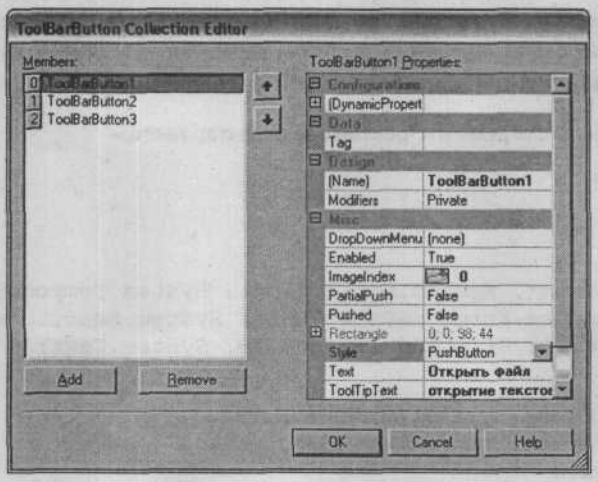

Рис. 5.18 ▾ Добавление кнопок к панели `ToolBar`

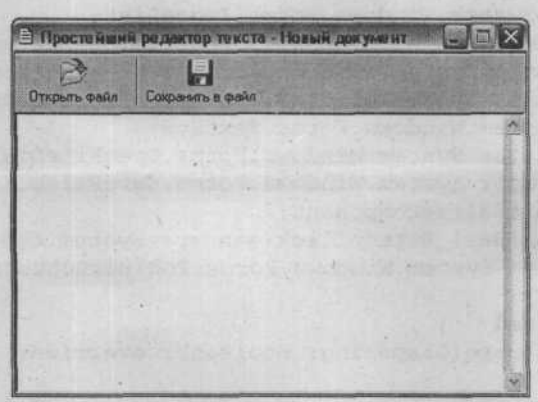

Рис. 5.19 ▾ Форма программы, использующей компонент `ToolBar`

Необходимо отметить некоторое отличие при использовании кнопок на панели ToolBar от рассмотренных нами ранее обычных кнопок Buttons. При нажатии на любую из кнопок панели инструментов возникает событие ButtonClick (не путайте с обычным событием Click, которое мы рассматривали раньше). При этом возникшее событие является *общим* для всех кнопок, расположенных на панели, соответственно, необходимо будет дополнительно узнавать, какая кнопка была нажата. Для этого воспользуйтесь функцией IndexOf, указав в качестве параметра коллекцию кнопок панели (см. ниже в тексте программы). Результатом выполнения этой функции будет индекс кнопки, которая была нажата (при этом необходимо помнить, что нумерация индексов начинается с нуля, а не с единицы).

Листинг 5.11 ▼ Текст программы «Простейший редактор текста»

```
unit WinForm;

interface

uses
   System.Drawing, System.Collections, System.ComponentModel,
   System.Windows.Forms, System.Data, System.Resources,
   System.Drawing.Printing, System.IO, System.Text;

type
 TWinForm = class(System.Windows.Forms.Form)
 {$REGION 'Designer Managed Code'}
 strict private
  components: System.ComponentModel.IContainer;
  ToolBar1: System.Windows.Forms.ToolBar;
  ImageList1: System.Windows.Forms.ImageList;
  ToolBarButton1: System.Windows.Forms.ToolBarButton;
  ToolBarButton2: System.Windows.Forms.ToolBarButton;
  ToolBarButton3: System.Windows.Forms.ToolBarButton;
  TextBox1: System.Windows.Forms.TextBox;
  OpenFileDialog1: System.Windows.Forms.OpenFileDialog;
  SaveFileDialog1: System.Windows.Forms.SaveFileDialog;
  procedure InitializeComponent;
  procedure ToolBar1_ButtonClick(sender: System.Object;
            e: System.Windows.Forms.ToolBarButtonClickEventArgs);
   {$ENDREGION}
 strict protected
  procedure Dispose(Disposing: Boolean); override;
 private
  file_name : string;          // Имя текстового файла.
  stream_read:System.IO.StreamReader;    // Поток для выполнения
                                         // чтения.
```

```
  stream_write:System.IO.StreamWriter;  // Поток для выполнения
                                        // записи.
  encoder:System.Text.Encoding;     // Кодировщик (необходим
                                    // для правильного отображения текста).
  procedure Open_File;    // Процедура для открытия текстового
                          // файла.
  procedure Save_File;    // Процедура для сохранения текстового
                          // файла.
 public
  constructor Create;
 end;

 [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm))]

implementation

{$AUTOBOX ON}

{$REGION 'Windows Form Designer generated code'}
procedure TWinForm.Dispose(Disposing: Boolean);
begin
 if Disposing then
 begin
  if Components <> nil then
   Components.Dispose();
 end;
 inherited Dispose(Disposing);
end;

constructor TWinForm.Create;
begin
 inherited Create;
 InitializeComponent;
 // Устанавливаем русскую кодировку для Windows (1251).
 encoder:=System.Text.Encoding.GetEncoding(1251);
end;

// Процедура обработки нажатия на кнопку панели инструментов.
procedure TWinForm.ToolBar1_ButtonClick(sender: System.Object;
            e: System.Windows.Forms.ToolBarButtonClickEventArgs);
begin
 // Определяем номер кнопки, которая была нажата.
 case (ToolBar1.Buttons.IndexOf(e.Button)) of
  0:  Open_File; // Была нажата кнопка 'Открыть файл' - вызов
                 // процедуры открытия файла Open_File
  // (кнопка с индексом 1 является разделителем).
  2:  Save_File;  // Была нажата кнопка 'Сохранить в файл' - вызов
```

```
                    // процедуры сохранения текста в файл Save_File.
 end;
end;

// Процедура открытия текстового файла
// (текст отображается в поле компонента TextBox1).
procedure TWinForm.Open_File;
begin
 OpenFileDialog1.FileName := '';
 // Отображаем диалоговое окно 'Открыть...'.
 OpenFileDialog1.ShowDialog;
 if OpenFileDialog1.FileName <> NIL then
 begin
   file_name := OpenFileDialog1.FileName;
   // Отображаем имя открытого файла в заголовке окна.
   Text := file_name;
   try
    stream_read := System.IO.StreamReader.Create(file_name,encoder);
    TextBox1.Text := stream_read.ReadToEnd;    // Читаем весь файл.
    stream_read.Close;                         // Закрываем поток.
    TextBox1.SelectionStart:=TextBox1.TextLength;
   except
    on e:exception do
     MessageBox.Show(e.Message);
   end;
 end;
end;

// Процедура сохранения текста в файл.
procedure TWinForm.Save_File;
begin
 // Если имя файла не задано, то выводим
 // диалоговое окно 'Сохранить...'.
 if file_name = NIL then
 begin
  SaveFileDialog1.ShowDialog;
  file_name := SaveFileDialog1.FileName;
 end;
 // В противном случае перезаписываем открытый файл.
  try
   // Открываем поток для перезаписи.
   stream_write:=System.IO.StreamWriter.Create(file_name,False,encoder);
   stream_write.Write(TextBox1.Text); // Записываем текст в поток.
```

```
    stream_write.Close;                    // Закрываем поток.
    self.Text := file_name;
   except
    on e : exception do
     MessageBox.Show(e.Message);
   end;
 end;
end.
```

Компонент ProgressBar

Этот компонент представляет собой индикатор, обычно используемый для отображения протекания какого-либо процесса. Такими процессами, например, могут быть копирование файлов, загрузка данных и т.п. Чтобы не раздражать людей, использующих ваши программы, кажущимся бездействием (хотя на самом деле выполняются какие-либо трудоемкие операции), можно использовать именно этот компонент. Свойства компонента представлены в табл. 5.20.

Таблица 5.20 ▾ Основные свойства компонента `ProgressBar`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Value`	Значение, которое отображается в поле компонента в виде прямоугольников, количество которых пропорционально значению свойства `Value`
`Minimum`	Минимальное значение, которое может принимать свойство `Value`
`Maximum`	Максимальное значение, которое может принимать свойство `Value`
`Step`	Свойство, определяющее приращение свойства `Value`. Используется для изменения свойства `Value` методом `PerformStep`

Обратите внимание на то, что при выходе значения свойства `Value` за границы диапазона, который определяется значениями `Minimum` и `Maximum`, возникнет ошибка.

В качестве примера напишем программу, которая копирует несколько файлов из одного каталога в другой. Форма окна программы приведена на рисунке 5.20, текст программы – в листинге 5.12.

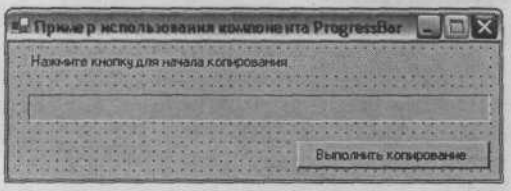

Рис. 5.20 ▾ Окно программы, использующей компонент `ProgressBar`

Листинг 5.12 ▼ Текст программы копирования файлов

```
unit WinForm1;

interface

uses
 System.Drawing, System.Collections, System.ComponentModel,
 System.Windows.Forms, System.Data,System.IO, Borland.Vcl.SysUtils;

type
 TWinForm1 = class(System.Windows.Forms.Form)
 {$REGION 'Designer Managed Code'}
 strict private
  Components: System.ComponentModel.Container;
  ProgressBar1: System.Windows.Forms.ProgressBar;
  Button1: System.Windows.Forms.Button;
  Label1: System.Windows.Forms.Label;
  procedure InitializeComponent;
  procedure Button1_Click(sender: System.Object;
                          e: System.EventArgs);
 {$ENDREGION}
 strict protected
  procedure Dispose(Disposing: Boolean); override;
 private
  dir_info:DirectoryInfo;  // Переменная для хранения информации
                           // о каталоге.
  files_info:array of FileInfo; // Массив для хранения информации
                                // о файлах.
  s:string;          // Переменная для определения текущего
                     // каталога.
 public
  constructor Create;
 end;

 [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm1))]

implementation

{$AUTOBOX ON}

{$REGION 'Windows Form Designer generated code'}
procedure TWinForm1.Dispose(Disposing: Boolean);
begin
 if Disposing then
 begin
  if Components <> nil then
   Components.Dispose();
 end;
```

```
 inherited Dispose(Disposing);
end;

constructor TWinForm1.Create;
begin
 inherited Create;
 InitializeComponent;
 s:=GetCurrentDir();
 try
   // Создаем объект DirectoryInfo.
  dir_info:=DirectoryInfo.Create(s+'\Folder_1');
 except
  on e:Exception do
   begin
    MessageBox.Show(e.Message);
   end;
 end;
 // Формируем список файлов (считываются файлы типа JPG).
 files_info:=dir_info.GetFiles('*.jpg');
 // Определаем максимальное значение свойства Value компонента
 // ProgressBar.
 ProgressBar1.Maximum:=Length(files_info);
end;
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
var i:integer;
begin
 // Делаем кнопку недоступной в момент копирования во избежание
 // ошибок.
 Button1.Enabled:=False;
 //Устанавливаем значение Value в 0.
 ProgressBar1.Value:=0;
 // Копируем поочередно все файлы.
 for i:=0 to ProgressBar1.Maximum-1 do
  begin
   Progressbar1.Increment(1);
   Label1.Text:='Выполняется копирование '+files_info[i].Name;
   Label1.Refresh;
   // Выполняем копирование в режиме перезаписи (True).
   files_info[i].CopyTo(s+'\Folder_2\'+files_info[i].Name,True);
 end;
  // Кнопка выполнения копирования снова доступна.
 Button1.Enabled:=True;
 Label1.Text:='Копирование завершено';
end;

end.
```

Программа работает следующим образом. Во время создания окна определяется текущий каталог, из которого была запущена программа. Затем в этом каталоге создается объект типа `DirectoryInfo`, который указывает на папку Folder_1. После этого формируется список файлов этого каталога (с помощью метода `GetFiles`) и в зависимости от их количества определяется значение свойства `Maximum` компонента `ProgressBar1`.

Далее после нажатия кнопки **Выполнить копирование** последовательно копируются файлы с одновременным изменением значения свойства `Value` (положением индикатора) компонента `ProgressBar1`.

Среди особенностей программы нужно отметить необходимость отключения кнопки `Button1` на весь период копирования во избежание возможных ошибок.

Компонент MainMenu

Этот компонент представляет собой главное меню программы. После добавления его к форме программы в верхней части окна появится строка, которую предстоит заполнить элементами меню. В начале строки находится область для ввода текста (прямоугольник с надписью **Type here**). Для создания элемента меню необходимо выполнить щелчок в области ввода текста и ввести название пункта меню. Как только вы что-нибудь наберете, справа и внизу появятся области для ввода следующих элементов меню. Заполняя эти области, вы постепенно создаете структуру вашего меню (рис. 5.21).

Пункты меню можно отделять друг от друга с помощью разделителей. Для этого необходимо навести курсор в то место, куда вы хотите поместить разделитель, и нажать правую кнопку мыши. В появившемся контекстном меню, выбрав пункт **Insert Separator** (Вставить разделитель), вы увидите, как появится полоска между пунктами меню. Также можно ввести символ «-» (минус), и Delphi сама преобразует этот пункт меню в разделитель.

Сформировав структуру, можно приступать к настройке главного меню. Для этого нам потребуется ознакомиться с основными свойствами компонента `MainMenu`, точнее говоря, со свойствами пунктов нашего меню (`MenuItem`).

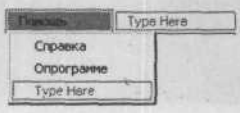

Рис. 5.21 ▼ Заполнение меню элементами

Доступ к свойствам элемента меню `MenuItem` можно получить из окна **Object Inspector**, нажав на один из пунктов меню, который мы собираемся настроить. Итак, основные свойства, которые нам предстоит настраивать, приведены в табл. 5.21.

Таблица 5.21 ▾ Основные свойства элементов меню `MainMenu`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Text`	Название элемента меню
`Enabled`	Признак доступности элемента меню. Если значение свойства равно `False`, то название пункта изображается инверсным цветом и при нажатии на него событие `Click` не происходит
`Checked`	Признак того, что элемент меню выбран. Выбранный элемент меню помечается галочкой
`ShortCut`	Свойство определяет комбинацию клавиш (или клавишу), нажатие которой расценивается как выбор соответствующего пункта меню
`ShowShortCut`	Свойство, определяющее, будет ли отображаться в пункте меню комбинация клавиш, нажатие которой активизирует данный пункт меню

Теперь поговорим о событиях, которые может воспринимать наше меню. Их два – `Click` (Нажатие) и `Select` (Выбор).

В первом случае событие возникает в результате щелчка по элементу меню, при нажатии клавиши **Enter** (если выбран пункт меню) или в результате нажатия функциональной клавиши, которая указана в свойстве `Shortcut` (клавиши быстрого доступа к этому элементу меню).

Второе же событие возникает при наведении указателя мыши на элемент меню, а также при его выборе с помощью клавиш управления курсором.

Следующая программа, которая является немного усовершенствованным редактором текстовых файлов, демонстрирует использование компонента `MainMenu`. Форма окна приведена на рис. 5.22. Текст программы приведен в листинге 5.13.

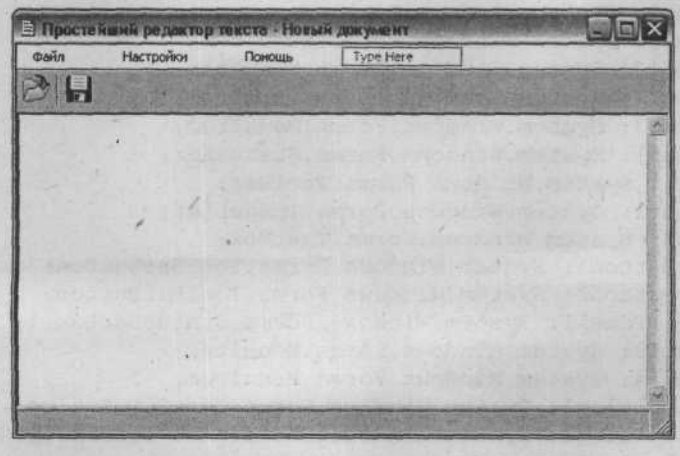

Рис. 5.22 ▾ Форма окна программы редактирования текстовых файлов

Во время работы программы в строке состояния отображается справка о выбранном пункте меню (с помощью процедуры `Select`). Выполнение же команд обеспечивает процедура `Click`.

Листинг 5.13 ▼ Текст программы «Простейший текстовый редактор» с использованием компонента `MainMenu`

```
unit WinForm;

interface

uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data, System.Resources,
  System.IO, System.Text;

type
  TWinForm = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    components: System.ComponentModel.IContainer;
    MainMenu1: System.Windows.Forms.MainMenu;
    MenuItem1: System.Windows.Forms.MenuItem;
    MenuItem2: System.Windows.Forms.MenuItem;
    MenuItem3: System.Windows.Forms.MenuItem;
    MenuItem4: System.Windows.Forms.MenuItem;
    MenuItem5: System.Windows.Forms.MenuItem;
    MenuItem6: System.Windows.Forms.MenuItem;
    MenuItem7: System.Windows.Forms.MenuItem;
    MenuItem9: System.Windows.Forms.MenuItem;
    MenuItem10: System.Windows.Forms.MenuItem;
    MenuItem11: System.Windows.Forms.MenuItem;
    MenuItem12: System.Windows.Forms.MenuItem;
    StatusBar1: System.Windows.Forms.StatusBar;
    ToolBar1: System.Windows.Forms.ToolBar;
    ImageList1: System.Windows.Forms.ImageList;
    TextBox1: System.Windows.Forms.TextBox;
    ToolBarButton1: System.Windows.Forms.ToolBarButton;
    ToolBarButton3: System.Windows.Forms.ToolBarButton;
    StatusBarPanel1: System.Windows.Forms.StatusBarPanel;
    MenuItem13: System.Windows.Forms.MenuItem;
    MenuItem14: System.Windows.Forms.MenuItem;
    OpenFileDialog1: System.Windows.Forms.OpenFileDialog;
    SaveFileDialog1: System.Windows.Forms.SaveFileDialog;
    FontDialog1: System.Windows.Forms.FontDialog;
    ToolBarButton2: System.Windows.Forms.ToolBarButton;
    procedure InitializeComponent;
```

```
 procedure MenuItem5_Select(sender: System.Object;
                            e: System.EventArgs);
 procedure MenuItem4_Select(sender: System.Object;
                            e: System.EventArgs);
 procedure MenuItem3_Select(sender: System.Object;
                            e: System.EventArgs);
 procedure MenuItem2_Select(sender: System.Object;
                            e: System.EventArgs);
 procedure MenuItem1_Select(sender: System.Object;
                            e: System.EventArgs);
 procedure MenuItem11_Click(sender: System.Object;
                            e: System.EventArgs);
 procedure MenuItem12_Click(sender: System.Object;
                            e: System.EventArgs);
 procedure MenuItem12_Select(sender: System.Object;
                             e: System.EventArgs);
 procedure MenuItem11_Select(sender: System.Object;
                             e: System.EventArgs);
 procedure MenuItem10_Select(sender: System.Object;
                             e: System.EventArgs);
 procedure MenuItem9_Select(sender: System.Object;
                            e: System.EventArgs);
 procedure MenuItem5_Click(sender: System.Object;
                           e: System.EventArgs);
 procedure ToolBar1_ButtonClick(sender: System.Object;
           e: System.Windows.Forms.ToolBarButtonClickEventArgs);
 procedure MenuItem3_Click(sender: System.Object;
                           e: System.EventArgs);
 procedure MenuItem4_Click(sender: System.Object;
                           e: System.EventArgs);
 procedure MenuItem2_Click(sender: System.Object;
                           e: System.EventArgs);
 procedure MenuItem10_Click(sender: System.Object;
                            e: System.EventArgs);
 procedure MenuItem7_Click(sender: System.Object;
                           e: System.EventArgs);
{$ENDREGION}
strict protected
 procedure Dispose(Disposing: Boolean); override;
private
 stream_read:System.IO.StreamReader; // Поток для выполнения
                                     // чтения.
 stream_write:System.IO.StreamWriter;   // Поток для выполнения
                                        // записи.
 encoder:System.Text.Encoding;     // Кодировщик.
```

```
  file_name:string;              // Переменная для хранения
                                 // имени текстового файла.
  procedure Open_File;           // Процедура открытия файла.
  procedure Save_File;           // Процедура сохранения файла
public
constructor Create;
 end;

 [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm))]

implementation

{$AUTOBOX ON}

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

constructor TWinForm.Create;
begin
 inherited Create;
 InitializeComponent;
 // Устанавливаем русскую кодировку для Windows (1251).
 encoder := System.Text.Encoding.GetEncoding(1251);
end;

// Выбор пункта меню 'О программе'.
procedure TWinForm.MenuItem7_Click(sender: System.Object;
                                   e: System.EventArgs);
begin
 MessageBox.Show('Простейший редактор текста с'+#13+
                 'использованием компонента MainMenu',
                 'О программе',
                 MessageBoxButtons.OK,
                 MessageBoxIcon.Information);
end;

// Выбор пункта меню 'Создать...'.
procedure TWinForm.MenuItem2_Click(sender: System.Object;
                                   e: System.EventArgs);
begin
 file_name := '';
 TextBox1.Text := '';
end;
```

```
// Выбор пункта меню 'Открыть файл...'.
procedure TWinForm.MenuItem3_Click(sender: System.Object;
                                   e: System.EventArgs);
begin
 Open_File;
end;

// Выбор пункта меню 'Сохранить в файл...'.
procedure TWinForm.MenuItem4_Click(sender: System.Object;
                                   e: System.EventArgs);
begin
 Save_File;
end;

// Выбор пункта меню 'Завершение работы...'.
procedure TWinForm.MenuItem5_Click(sender: System.Object;
                                   e: System.EventArgs);
begin
 Close;
end;

// Выбор пункта меню 'Настройки/Выбор Шрифта'.
procedure TWinForm.MenuItem10_Click(sender: System.Object;
                                    e: System.EventArgs);
var
 d_result:System.Windows.Forms.DialogResult;
begin
 d_result:=FontDialog1.ShowDialog;
 if (d_result=System.Windows.Forms.DialogResult.OK) then
 begin
  TextBox1.Font := FontDialog1.Font;
 end;
end;

// Выбор пункта меню 'Настройки/Панель инструментов'.
procedure TWinForm.MenuItem11_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 ToolBar1.Visible := not ToolBar1.Visible;
 MenuItem11.Checked := not MenuItem11.Checked;
end;
// Выбор пункта меню 'Настройки/Строка состояния'.
procedure TWinForm.MenuItem12_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 // Скрыть/показать строку состояния.
 StatusBar1.Visible := not StatusBar1.Visible;
```

```
 // Установить/сбросить флажок рядом с командой.
 MenuItem12.Checked := not MenuItem12.Checked;
 if StatusBar1.Visible then StatusBar1.Panels[0].Text :='';
end;

// Щелчок на кнопке панели инструментов.
procedure TWinForm.ToolBar1_ButtonClick(sender: System.Object;
            e: System.Windows.Forms.ToolBarButtonClickEventArgs);
begin
 // Определяем кнопку, которая была нажата.
 case (ToolBar1.Buttons.IndexOf(e.Button)) of
  0:  Open_File; // Открытие текстового файла
                 // кнопка с индексом 1 - разделитель.
  2:  Save_File; // Сохранение в файл.
 end;
end;

// Мышь над пунктом меню 'Настройки'.
procedure TWinForm.MenuItem9_Select(sender: System.Object;
                                    e: System.EventArgs);
begin
  StatusBar1.Panels[0].Text := '';
end;

// Мышь над пунктом меню 'Настройки/Шрифт'.
procedure TWinForm.MenuItem10_Select(sender: System.Object;
                                     e: System.EventArgs);
begin
 StatusBar1.Panels[0].Text := 'Выбор шрифта';
end;

// Мышь над пунктом меню 'Настройки/Панель инструментов'.
procedure TWinForm.MenuItem11_Select(sender: System.Object;
                                     e: System.EventArgs);
begin
 if MenuItem11.Checked
  then
   StatusBar1.Panels[0].Text := 'Скрыть панель инструментов'
  else
   StatusBar1.Panels[0].Text := 'Показать панель инструментов';
end;

// Мышь над пунктом меню 'Настройки/Строка состояния'.
procedure TWinForm.MenuItem12_Select(sender: System.Object;
                                     e: System.EventArgs);
begin
  if MenuItem11.Checked
```

```
   then StatusBar1.Panels[0].Text := 'Скрыть строку состояния'
   else StatusBar1.Panels[0].Text := 'Показать строку состояния';
end;

// Мышь над пунктом меню 'Файл'.
procedure TWinForm.MenuItem1_Select(sender: System.Object;
                                    e: System.EventArgs);
begin
 StatusBar1.Panels[0].Text := '';
end;

// Мышь над пунктом меню 'Файл/Создать...'.
procedure TWinForm.MenuItem2_Select(sender: System.Object;
                                    e: System.EventArgs);
begin
 StatusBar1.Panels[0].Text := 'Начало работы над новым файлом';
end;

// Мышь над пукнтом меню команде 'Файл/Открыть файл'.
procedure TWinForm.MenuItem3_Select(sender: System.Object;
                                    e: System.EventArgs);
begin
 StatusBar1.Panels[0].Text := 'Открыть существующий файл';
end;

// Мышь над пунктом меню 'Файл/Сохранить в файл'.
procedure TWinForm.MenuItem4_Select(sender: System.Object;
                                    e: System.EventArgs);
begin
 StatusBar1.Panels[0].Text := 'Сохранить текст в файле';
end;

// Мышь над пунктом меню 'Файл/Завершение работы'.
procedure TWinForm.MenuItem5_Select(sender: System.Object;
                                    e: System.EventArgs);
begin
 StatusBar1.Panels[0].Text := 'Завершить работу с программой';
end;

// Открытие файла и отображение его в поле
// компонента TextBox1.
procedure TWinForm.Open_File;
begin
 OpenFileDialog1.FileName := '';
 // Отображение диалогового окна 'Открыть...'.
 OpenFileDialog1.ShowDialog;
 if OpenFileDialog1.FileName <> NIL then
 begin
```

```
    file_name := OpenFileDialog1.FileName;
    // Отображение имени файла в заголовке окна.
    Text := file_name;
    try
     stream_read:=System.IO.StreamReader.Create(file_name,encoder);
       TextBox1.Text := stream_read.ReadToEnd; // Читаем весь файл.
       stream_read.Close;
       TextBox1.SelectionStart:=TextBox1.TextLength;
    except
     on e:exception do
      MessageBox.Show(e.Message);
    end;
  end;
end;

// Сохранение текста в файле.
procedure TWinForm.Save_File;
begin
 // Если имя файла не задано, то выводим
 // диалоговое окно 'Сохранить...'.
 if file_name = NIL then
 begin
  SaveFileDialog1.ShowDialog;
   file_name := SaveFileDialog1.FileName;
   // Открываем поток для перезаписи.
   stream_write:=System.IO.StreamWriter.Create(file_name,False,encoder);
   stream_write.Write(TextBox1.Text); // Записываем текст в поток.
   stream_write.Close;                // Закрываем поток.
   Self.Text:=file_name;
 end
 // В противном случае
 else
  try
   // открываем поток для перезаписи.
   stream_write:=System.IO.StreamWriter.Create(file_name,False,encoder);
    stream_write.Write(TextBox1.Text); // Записываем текст в поток.
    stream_write.Close;                // Закрываем поток.
    Self.Text:=file_name;
  except
   on e : exception do
    MessageBox.Show(e.Message);
  end;
end;

end.
```

Компонент ContextMenu

Этот компонент также представляет собой меню. От предыдущего он отличается тем, что пункты этого меню отображаются при нажатии *правой* кнопки мыши (такое меню называется *контекстным*). После добавления этого компонента на форму приложения в строке свойств формы появится новое свойство – `ContextMenu`. Для определения перечня пунктов меню нужно выполнить двойной щелчок по этому свойству и заполнить элементы меню так же, как мы делали это для главного меню.

После того как контекстное меню будет создано, следует выполнить его окончательную настройку – задать значения свойств пунктов меню `MenuItem`, а также определить процедуры обработки событий. В отличие от компонента `MainMenu` для `ContextMenu` необходимо *дополнительно* определить компонент, для которого это меню создано. Для этого в свойство `ContextMenu` компонента необходимо поместить ссылку на контекстное меню. Свойства объекта `MenuItem` приведены в табл. 5.22.

Таблица 5.22 ▾ Основные свойства элемента меню `ContextMenu`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Text`	Название элемента меню
`Enabled`	Признак доступности элемента меню. Если значение свойства равно `False`, то название пункта изображается инверсным цветом и при нажатии на него событие `Click` не происходит
`Checked`	Признак того, что элемент меню выбран. Выбранный элемент меню помечается галочкой
`RadioCheck`	Признак того, что для пометки пункта меню используется точка, а не галочка

Следующая программа демонстрирует использование компонента `ContextMenu`. Форма окна программы приведена на рис. 5.23.

Текст программы приведен в листинге 5.14.

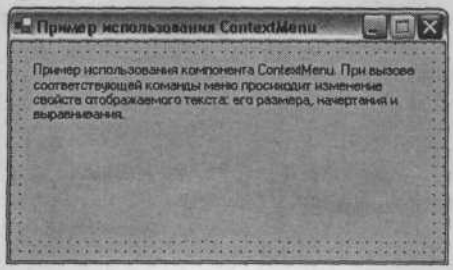

Рис. 5.23 ▾ Форма программы, демонстрирующей вариант использования компонента `ContextMenu`

Листинг 5.14 ▼ Текст программы (основные процедуры), демонстрирующей вариант использования компонента ContextMenu.

```
// Выбор пункта меню 'Выравнивание текста/По центру'.
procedure TWinForm1.MenuItem9_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 MenuItem7.Checked:=True;
 MenuItem8.Checked:=False;
 MenuItem9.Checked:=False;
 Label1.TextAlign:=System.Drawing.ContentAlignment.TopCenter;
end;

// Выбор пункта меню 'Выравнивание текста/По правому краю'.
procedure TWinForm1.MenuItem8_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 MenuItem7.Checked:=False;
 MenuItem8.Checked:=True;
 MenuItem9.Checked:=False;
 Label1.TextAlign:=System.Drawing.ContentAlignment.TopRight;
end;

// Выбор пункта меню 'Выравнивание текста/По левому краю'.
procedure TWinForm1.MenuItem7_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 MenuItem7.Checked:=True;
 MenuItem8.Checked:=False;
 MenuItem9.Checked:=False;
 Label1.TextAlign:=System.Drawing.ContentAlignment.TopLeft;
end;
// Выбор пункта меню 'Выбор шрифта'.
procedure TWinForm1.MenuItem2_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 FontDialog1.ShowDialog;
 Label1.Font:=FontDialog1.Font;
end;

// Выбор пункта меню 'Размер шрифта/Крупный'.
procedure TWinForm1.MenuItem6_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 MenuItem4.Checked:=False;
 MenuItem5.Checked:=False;
 MenuItem6.Checked:=True;
```

```
 Label1.Font:=System.Drawing.Font.Create(Label1.Font.FontFamily,12);
end;

// Выбор пункта меню 'Размер шрифта/Средний'.
procedure TWinForm1.MenuItem5_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 MenuItem4.Checked:=False;
 MenuItem5.Checked:=True;
 MenuItem6.Checked:=False;
 Label1.Font:=System.Drawing.Font.Create(Label1.Font.FontFamily,10);
end;

// Выбор пункта меню 'Размер шрифта/Мелкий'.
procedure TWinForm1.MenuItem4_Click(sender: System.Object;
                                    e: System.EventArgs);
begin
 MenuItem4.Checked:=True;
 MenuItem5.Checked:=False;
 MenuItem6.Checked:=False;
 Label1.Font:=System.Drawing.Font.Create(Label1.Font.FontFamily,8);
end;
```

Компонент OpenFileDialog

С компонентом `OpenFileDialog` мы уже сталкивались, когда создавали простейший текстовый редактор. Данный компонент представляет собой стандартное диалоговое окно, позволяющее выбирать (открывать) файлы. Хотя некоторые свойства этого компонента нам уже знакомы, считаю необходимым привести основные из них (табл. 5.23).

Таблица 5.23 ▾ Основные свойства компонента `OpenFileDialog`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Title`	Заголовок диалогового окна
`Filename`	Имя выбранного пользователем файла. Помимо собственно имени файла содержит также и путь к каталогу, в котором находится файл
`InitialDirectory`	Имя каталога, содержимое которого будет отображаться при появлении окна
`RestoreDirectory`	Признак необходимости того, что содержимое каталога, указанное в свойстве `InitialDirectory`, будет отображаться при каждом появлении окна на экране. Если значение свойства установлено в `False`, то при следующем появлении окна на экране отобразится каталог, который был выбран в прошлый раз

Таблица 5.23 ▼ Основные свойства компонента `OpenFileDialog` (окончание)

Свойство	Комментарий
`Filter`	Свойство определяет описание и фильтр, по которому будут отображаться файлы. В списке файлов отображаются только те файлы, описание которых соответствует заданной маске. Например, значение свойства `Текстовые файлы | *.txt` будет отображать только текстовые файлы с соответствующим расширением
`FilterIndex`	Если фильтр имеет несколько элементов (например, `Текстовые файлы | *.txt | Файлы изображений |*.jpg`), то значение свойства определяет тот фильтр, который используется в момент появления окна на экране

Отображение самого диалогового окна обеспечивает метод `ShowDialog`. Результатом завершения будет код клавиши, которая была нажата пользователем (**OK** или **Cancel**). Если была нажата клавиша **OK**, то результатом выполнения метода будет `DialogResult.Ok`. Помните, что данный компонент *не выполняет* непосредственно открытия файла, – его назначение состоит в том, чтобы получить имя файла, над которым будут производиться соответствующие действия (например, чтение файла).

Следующая программа использует компонент `OpenFileDialog` для открытия текстового файла, который затем отображается с помощью компонента `TextBox`. Форма окна программы приведена на рис. 5.24, текст программы – в листинге 5.15.

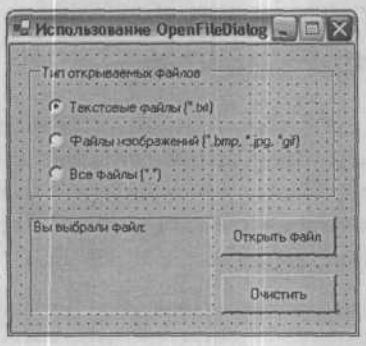

Рис. 5.24 ▼ Форма программы, использующей компонент `OpenFileDialog`

Листинг 5.15 ▼ Текст программы (основные процедуры), демонстрирующей использование компонента `OpenFileDialog`

```
// Нажатие на кнопку 'Очистить'.
procedure TWinForm1.Button2_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
  OpenFileDialog1.FileName:='';
  Label1.Text:='Вы выбрали файл: ';
```

```
end;
// Нажатие на кнопку 'Открыть файл'.
procedure TWinForm1.Button1_Click(sender: System.Object;
                                  e: System.EventArgs);
begin
 // Определяем текущий каталог.
 OpenFileDialog1.InitialDirectory:=Environment.CurrentDirectory;
 // Определяем, какой фильтр отображения применять.
 if RadioButton1.Checked then
  OpenFileDialog1.Filter:='Текстовые файлы|*.txt'; // Простой
                                                   // фильтр.
 if RadioButton2.Checked then
 // Составной фильтр.
  OpenFileDialog1.Filter:='Файлы изображений|*.bmp;*.jpg;*.gif';
 if RadioButton3.Checked then
  OpenFileDialog1.Filter:='Все файлы|*.*'; // Простой фильтр - все
                                           // файлы.
 OpenFileDialog1.ShowDialog;
 Label1.Text:='Вы выбрали файл: '+OpenFileDialog1.FileName;
end;
```

Диалоговое окно открытия файла появляется в результате нажатия кнопки **Открыть файл**. При этом фильтр отображения файлов формируется в зависимости от выбранного значения в группе **Тип открываемых файлов**. В начале работы программы в диалоге текущим будет каталог, из которого была запущена программа (имя текущего каталога содержит свойство `CurrentDirectory` объекта `Environment`).

Отображение окна открытия файла, чтение и отображение текстового файла осуществляет метод `ShowDialog` компонента `OpenFileDialog1`.

Компонент SaveFileDialog

Этот компонент, также предназначенный для работы с именами файлов, служит для сохранения файлов. Свойства компонента приведены в табл. 5.24.

Таблица 5.24 ▾ Основные свойства компонента `SaveFileDialog`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Title`	Заголовок диалогового окна. Если свойство не указано, то заголовок будет иметь стандартное значение **Сохранить как**

Таблица 5.24 ▼ Основные свойства компонента `SaveFileDialog` (окончание)

Свойство	Комментарий
`Filename`	Имя выбранного пользователем файла. Помимо собственно имени содержит так же и путь к каталогу, в котором находится файл
`InitialDirectory`	Имя каталога, содержимое которого будет отображаться при появлении окна
`RestoreDirectory`	Признак необходимости того, что содержимое каталога, указанное в свойстве `InitialDirectory`, будет отображаться при каждом появлении окна на экране. Если значение свойства установлено в `False`, то при следующем появлении окна на экране отобразится каталог, который был выбран в прошлый раз
`Filter`	Свойство определяет описание и фильтр, по которому будут отображаться файлы. В списке файлов отображаются только те файлы, описание которых соответствует заданной маске. Например, значение свойства `Текстовые файлы | *.txt` будет отображать только текстовые файлы с соответствующим расширением
`FilterIndex`	Если фильтр имеет несколько элементов (например, `Текстовые файлы | *.txt | Файлы изображений |*.jpg`), то значение свойства определяет тот фильтр, который используется в момент появления окна на экране
`DefaultExt`	Расширение файла, задаваемое по умолчанию (если оно не введено при сохранении). Данное расширение добавляется к имени файла в случае, если свойство `AddExtension` установлено в `True`
`AddExtension`	Признак необходимости добавлять расширение (в случае, если пользователь не ввел его при сохранении)
`CheckPathExists`	Признак необходимости проверки существования каталога, в котором будет сохранен файл. Если искомый каталог не будет найден, то будет выведено информационное сообщение
`CheckFileExists`	Признак необходимости проверки существования файла с указанным именем. Если свойство установлено в `True` и указанный файл существует, то будет выведено сообщение, в котором предлагается подтвердить необходимость замены существующего файла

Отображение диалогового окна сохранения файла обеспечивает уже знакомый нам метод `ShowDialog`, значением которого является код кнопки, нажатием которой пользователь завершил диалог (закрыл окно). Приведенная ниже программа демонстрирует основные возможности этого компонента.

При успешном завершении диалога (нажатии кнопки **OK**) формируется текстовый файл, в который записывается содержимое компонента `TextBox` (рис. 5.25).

Текст программы (процедура обработки события `Click` на кнопке **Сохранить в файл**) приведен в листинге 5.16.

Листинг 5.16 ▼ Текст программы, демонстрирующей использование компонента `SaveFileDialog`

```
unit WinForm1;

interface

uses
  System.Drawing, System.Collections, System.ComponentModel,
```

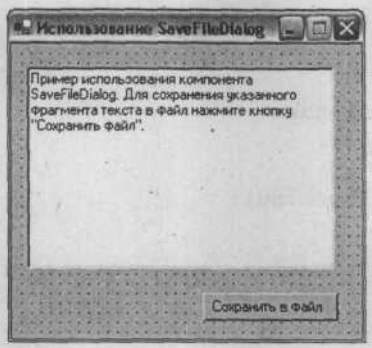

Рис. 5.25 ▾ Форма окна программы, демонстрирующей использование SaveFileDialog

```
 System.Windows.Forms, System.Data, System.IO, System.Text;
type
 TWinForm1 = class(System.Windows.Forms.Form)
 {$REGION 'Designer Managed Code'}
 strict private
  Components: System.ComponentModel.Container;
  SaveFileDialog1: System.Windows.Forms.SaveFileDialog;
  TextBox1: System.Windows.Forms.TextBox;
  Button1: System.Windows.Forms.Button;
  procedure InitializeComponent;
  procedure Button1_Click(sender: System.Object;
                          e: System.EventArgs);
 {$ENDREGION}
 strict protected
  procedure Dispose(Disposing: Boolean); override;
 private
  stream_write:System.IO.StreamWriter;  // Поток для осуществления
                                        // записи.
  encoder:System.Text.Encoding;
 public
  constructor Create;
 end;
 [assembly: RuntimeRequiredAttribute(TypeOf(TWinForm1))]
implementation
{$AUTOBOX ON}
{$REGION 'Windows Form Designer generated code'}
procedure TWinForm1.Dispose(Disposing: Boolean);
```

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

constructor TWinForm1.Create;
begin
 inherited Create;
 InitializeComponent;
 encoder:=System.Text.Encoding.GetEncoding(1251);
end;

// Нажатие на кнопку 'Сохранить в файл'.
procedure TWinForm1.Button1_Click(sender: System.Object;
    e: System.EventArgs);
begin
 // Если пользователь закрыл диалог нажатием кнопки OK,
 if SaveFileDialog1.ShowDialog=System.Windows.Forms.DialogResult.OK
  then
   // то записать текст в файл.
   try
     stream_write:=

System.IO.StreamWriter.Create(SaveFileDialog1.FileName,False,encoder);
     stream_write.Write(TextBox1.Text); // Запись текста в поток.
     stream_write.Close;          // Закрыть поток.
   except
    on e:exception do
     MessageBox.Show(e.Message);
   end;
end;
end.
```

Изучаем основы работы с графикой в .NET

Эта глава посвящена описанию графических возможностей Delphi. Мы уже касались вопросов отображения картинок в наших программах, но это было лишь первое знакомство. Далее вы узнаете, как сформировать картинку самостоятельно или же использовать графические файлы для того, чтобы приукрасить вашу программу.

Создавая программу, вы можете использовать специальный компонент, отображающий картинки, – `PictureBox`, а также рисовать непосредственно на поверхности формы. Поверхности формы соответствует объект `Graphics`, методы которого и обеспечивают вывод графики. В любом случае, отображаете ли вы графический файл или рисуете самостоятельно, вам необходимо будет использовать объект `Graphics` (то есть применить соответствующие методы). Рисовать в Delphi очень просто – достаточно написать процедуру обработки события `Paint` (Рисование). В качестве примера приведу программу, которая обрабатывает это событие и выводит на поверхность формы картинку, а также простейшие графические примитивы (рис. 6.1).

Текст процедуры обработки события `Paint` приведен ниже в листинге 6.1.

Листинг 6.1 ▾ Процедура обработки события `Paint`

```
procedure TWinForm1.TWinForm1_Paint(sender: System.Object;
                        e: System.Windows.Forms.PaintEventArgs);
var
  my_image:Image;                  // Картинка.
  my_brush:Brush;                  // Кисть.
  my_font:System.Drawing.Font;     // Шрифт.
  my_pen:Pen;                      // Карандаш.
```

Рис. 6.1 ▼ Пример использования графических возможностей

```
begin
  // Отображение логотипа .NET.
  // Загрузка изображения из файла в my_image.
  my_image:=Image.FromFile('dotNET.bmp');
  // Отображение картинки на графической поверхности.
  e.Graphics.DrawImage(my_image,point.Create(10,10));
  // Отображение надписи с помощью графики.
  // Создаем кисть.
  my_brush:=SolidBrush.Create(Color.get_Blue);
  my_font:=System.Drawing.Font.Create('Arial',14);
  e.Graphics.DrawString('Графические возможности',
    my_font,my_brush,140,10);
  e.Graphics.DrawString('Delphi for .NET',
    my_font,my_brush,140,40);
  // Отображаем графические примитивы.
  // Настраиваем карандаш.
  my_pen:=Pen.Create(Color.get_Brown);
  // Рисуем линию.
  e.Graphics.DrawLine(my_pen,10,80,380,80);
  // Рисуем прямоугольные области.
  e.Graphics.DrawRectangle(my_pen,10,100,100,40);
  e.Graphics.DrawRectangle(my_pen,15,110,100,40);
  e.Graphics.DrawRectangle(my_pen,20,120,100,40);
  // Рисуем закрашенные прямоугольные области.
  e.Graphics.FillRectangle(my_brush,180,100,100,40);
  e.Graphics.FillRectangle(Brushes.Red,185,110,100,40);
  e.Graphics.FillRectangle(Brushes.Green,190,120,100,40);
end;
```

Несколько слов о событии `Paint`. Оно возникает всякий раз, когда появляется необходимость в перерисовке окна приложения (например, когда ваше окно было закрыто другим окном, а потом было сделано активным).

Теперь немного об особенностях методов, использующих окно формы для рисования. Эти методы используют так называемую *графическую* поверхность окна, которая состоит из точек (пикселей). Положение каждой точки характеризуется двумя координатами: горизонтальной (X) и вертикальной (Y). При этом также необходимо учесть, что координаты отсчитываются несколько необычно – от левого верхнего угла (рис. 6.2). Максимальный размер, который может быть использован для рисования, хранит свойство `ClientsSize` формы.

Далее познакомимся с основными инструментами, позволяющими рисовать на поверхности формы.

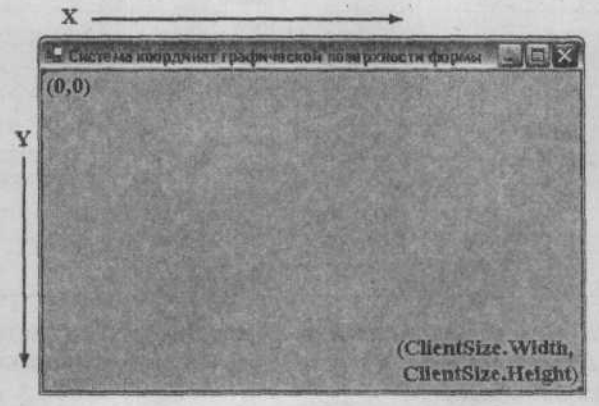

Рис. 6.2 ▾ Система координат точек графической поверхности формы

Карандаш и кисть – основные инструменты для рисования

Рисование осуществляется специальными инструментами – карандашами или кистями. Методы, позволяющие рисовать, используют именно эти инструменты. Карандаш позволяет задать вид линии (толщину, цвет и стиль), а кисть – способ закраски определенной области.

Карандаш

Объект «карандаш» (Pen) определяет вид линии – толщину, цвет и стиль. В нашем распоряжении находится два вида карандашей – *стандартные* и *системные*.

Стандартный набор карандашей – четырнадцать цветных карандашей, рисующих непрерывную линию толщиной в один пиксель. Некоторые карандаши из стандартного набора приведены в табл. 6.1. Если у вас есть желание увидеть полный набор стандартных карандашей, то можете воспользоваться справочной системой. Для этого в поле **Look for** вкладки **Index** введите **Pens**, затем в списке выберите **All members**.

Таблица 6.1 ▼ Основные карандаши из системного набора

Тип карандаша	Цвет
`Pens.Black`	Черный
`Pens.White`	Белый
`Pens.LightGray`	Серый
`Pens.Yellow`	Желтый
`Pens.Orange`	Оранжевый
`Pens.Red`	Красный
`Pens.Green`	Зеленый
`Pens.Purple`	Фиолетовый
`Pens.LightBlue`	Голубой
`Pens.Blue`	Синий
`Pens.Transparent`	Прозрачный

Ниже приведен пример использования карандашей из системного набора – процедура обработки события `Paint`, которая рисует несколько фигур с использованием карандашей.

```
procedure TWinForm.TWinForm_Paint(sender:System.Object;
                                  e: System.Windows.Forms.PaintEventArgs);
begin
  e.Graphics.DrawEllipse(Pens.Blue,20,20,20,20);
  e.Graphics.DrawRectangle(Pens.Yellow,100,100,20,20);
  e.Graphics.DrawEllipse(Pens.Green,220,20,30,30);
  e.Graphics.DrawRectangle(Pens.Red,90,40,100,50);
end;
```

Теперь перейдем к рассмотрению системного набора карандашей. Системный набор отличается от стандартного тем, что цвет карандашей «привязан» к цветовой схеме операционной системы. Например, установив цвет карандашей в `SystemPens.WindowText`, мы увидим цвет, который совпадает с цветом текста в окнах сообщений.

Все вышеперечисленные наборы карандашей рисуют непрерывную линию толщиной в *один* пиксель. Если же необходимо нарисовать пунктирную линию или линию, толщина которой больше одного пикселя, то следует создать и использовать так называемый карандаш программиста.

Карандаш программиста – объект типа `Pen`, имеющий свойства, определяющие вид линии, которую чертит карандаш. Ниже в табл. 6.2 приведены свойства объекта `Pen`.

Таблица 6.2 ▾ Свойства объекта `Pen`

Свойство карандаша	Комментарий
`Color`	Определяет цвет выводимой линии
`Width`	Толщина линии, задаваемая в пикселях
`DashStyle`	Свойство задает тип выводимой линии. Линия может быть сплошной (`DashStyle.Solid`), пунктирной (`DashStyle.Dash`), пунктирной с короткими штрихами (`DashStyle.Dot`), пунктирной с чередованием длинных и коротких штрихов (`DashStyle.DashDot`) и линией, вид которой определяется свойством `DashPattern` (`DashStyle.Custom`)
`DashPattern`	Свойство определяет длину штрихов линии, если она имеет тип `DashStyle.Custom`

Важно помнить, что перед тем как использовать свой карандаш, его необходимо создать. Для создания собственного карандаша необходимо воспользоваться методом `Create`. В качестве параметров этого метода используются константа типа `Color` (цвет линии) и толщина линии. По ходу выполнения программы можно менять свойства созданного карандаша, что приведет к изменению вида вычерчиваемых линий. Примеры констант типа `Color`, определяющих цвет карандаша, приведены в табл. 6.3.

Таблица 6.3 ▾ Константы типа `Color`

Константа	Цвет
`Color.Black`	Черный
`Color.White`	Белый
`Color.LightGray`	Серый
`Color.Yellow`	Желтый
`Color.Orange`	Оранжевый
`Color.Red`	Красный
`Color.Green`	Зеленый
`Color.Purple`	Фиолетовый
`Color.LightBlue`	Голубой
`Color.Blue`	Синий
`Color.Transparent`	Прозрачный

Ниже в листинге 6.2 приведен фрагмент программы (процедура обработки события `Paint`), которая демонстрирует создание и использование собственного карандаша[1].

[1] Для корректной компиляции этого примера укажите в списке включаемых модулей (**uses**) модуль `System.Drawing.Drawing2D`. – *Прим. науч. ред.*

Листинг 6.2 ▼ Создание и использование собственного карандаша

```
procedure TWinForm.TWinForm_Paint(sender: System.Object;
                                  e: System.Windows.Forms.PaintEventArgs);
var
  My_Pen : Pen;
begin
  // Создаем карандаш.
  My_Pen := Pen.Create(Color.Gray,5);
  // Рисуем квадрат.
  e.Graphics.DrawRectangle(My_Pen,80,10,50,50);
  // Изменяем свойства карандаша.
  My_Pen.Color := Color.Blue;
  My_Pen.Width := 3;
  My_Pen.DashStyle := DashStyle.Dot;
  // Рисуем прямоугольник точками.
  e.Graphics.DrawRectangle(My_Pen,30,40,80,60);
  // Изменяем свойства карандаша.
  My_Pen.Color := Color.Green;
  My_Pen.Width := 5;
  My_Pen.DashStyle := DashStyle.Solid;
  // Рисуем эллипс.
  e.Graphics.DrawEllipse(My_Pen,80,80,100,40);
  // Изменяем свойства карандаша.
  My_Pen.Color := Color.Black;
  My_Pen.Width :=2;
  My_Pen.DashStyle := DashStyle.Dash;
  // Рисуем эллипс пунктиром.
  e.Graphics.DrawEllipse(My_Pen,200,10,80,120);
end;
```

Результат работы программы показан на рис. 6.3.

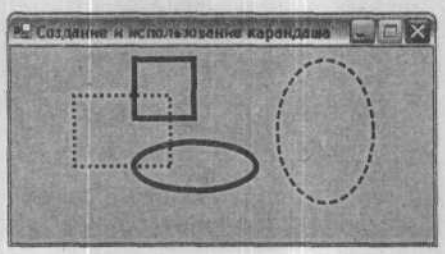

Рис. 6.3 ▼ Рисование с использованием собственного карандаша

Кисть

Инструмент «кисть» используется для закраски внутренних областей геометрических фигур. Например, инструкция следующего вида

```
e.Graphics.FillRectangle(Brushes.Orange,x,y,_width,_height)
```

рисует закрашенный прямоугольник.

В приведенном примере `Brushes.Orange` – стандартная кисть оранжевого цвета. Соответственно, параметры `x` и `y` определяют положение прямоугольника на экране, а параметры `_width` и `_height` – его размер.

Программист может использовать для своих целей четыре типа кистей:

- *стандартные.* Стандартная кисть закрашивает область одним цветом, при этом возможно использование 14 цветов, некоторые из которых приведены в табл. 6.4;
- *штриховые.* Этот тип кисти (`HatchBrush`) закрашивает область путем штриховки. Область может быть заштрихована горизонтальными, вертикальными или наклонными линиями. Некоторые из возможных стилей штриховки приведены в табл. 6.5;
- *текстурные.* Текстурная кисть (`TextureBrush`) использует для закраски области рисунок, который обычно загружается из файла (bmp, jpg и т.п.). При таком типе закраски область заполняется дублированием рисунка;
- *градиентные.* Данная кисть (`LinearGradientBrush`) представляет собой прямоугольную область, цвет точек в которой зависит от расстояния до границы. Такие кисти обычно являются двухцветными, то есть цвет точек в них плавно меняется с одного на другой.

Таблица 6.4 ▾ Основные кисти из стандартного набора

Кисть	Цвет
`Brushes.Black`	Черный
`Brushes.White`	Белый
`Brushes.LightGray`	Серый
`Brushes.Yellow`	Желтый
`Brushes.Orange`	Оранжевый
`Brushes.Red`	Красный
`Brushes.Green`	Зеленый
`Brushes.Purple`	Фиолетовый
`Brushes.LightBlue`	Голубой
`Brushes.Blue`	Синий
`Brushes.Transparent`	Прозрачный

Таблица 6.5 ▼ *Основные типы штриховки областей*

Стиль	Штриховка
HatchStyle.NarrowHorizontal	Частая горизонтальная
HatchStyle.Horizontal	Средняя горизонтальная
HatchStyle.LightHorizontal	Редкая горизонтальная
HatchStyle.NarrowVertical	Частая вертикальная
HatchStyle.Vertical	Вертикальная
HatchStyle.LightVertical	Редкая вертикальная
HatchStyle.LargeGrid	Крупная сетка
HatchStyle.SmallGrid	Мелкая сетка
HatchStyle.DottedGrid	Сетка, составленная из точек
HatchStyle.Percent05 - HatchStyle.Percent90	Точки со степенью заполнения 5–90%
HatchStyle.HorizontalBrick	Штриховка, напоминающая кирпичную стенку
HatchStyle.LargeCheckerBoard	Штриховка «шахматная доска»
HatchStyle.Sphere	Штриховка «пузырьки»

Приведем несколько примеров использования кистей. В качестве примера использования штриховой кисти приведен листинг 6.3, где демонстрируется процесс создания и использования штриховой кисти. Кисть создается методом `Create`. В качестве параметра этому методу передается константа `HatchStyle`, задающая тип штриховки, а также две константы типа `Color`, задающие цвет фона и штрихов соответственно. Результат работы программы приведен на рис. 6.4.

Листинг 6.3 ▼ Пример создания и использования штриховой кисти

```
unit WinForm;

interface

uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data,
  System.Drawing.Drawing2D;

type
  TWinForm = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    Components: System.ComponentModel.Container;
    procedure InitializeComponent;
    procedure TWinForm_Paint(sender: System.Object;
                             e: System.Windows.Forms.PaintEventArgs);
   {$ENDREGION}
  strict protected
    procedure Dispose(Disposing: Boolean); override;
```

```
    procedure Draw(g: Graphics; var r: Rectangle;
                   aHatchStyle: System.Drawing.Drawing2D.HatchStyle);
  public
    constructor Create;
  end;

implementation

uses
  System.Globalization;

{$REGION 'Windows Form Designer generated code'}
var
  rec:Rectangle;
  My_HatchBrush: HatchBrush;

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

constructor TWinForm.Create;
begin
  inherited Create;
  InitializeComponent;
  rec.X := 0;
  rec.Y := 0;
  rec.Width := 80;
  rec.Height :=40;
  Width := rec.Width *8 + 7;
  My_HatchBrush:=nil;
end;

procedure TWinForm.TWinForm_Paint(sender: System.Object;
                                  e: System.Windows.Forms.PaintEventArgs);
begin
  rec.X := 0;
  rec.Y := 0;
  // Отображаем возможные штриховки
  // (последовательно вызываем процедуру отображения текущей
  // штриховки Draw).
  Draw(e.Graphics,rec,HatchStyle.BackwardDiagonal);
  Draw(e.Graphics,rec,HatchStyle.Cross);
```

```
Draw(e.Graphics,rec,HatchStyle.DarkDownwardDiagonal);
Draw(e.Graphics,rec,HatchStyle.DarkHorizontal);
Draw(e.Graphics,rec,HatchStyle.DarkUpwardDiagonal);
Draw(e.Graphics,rec,HatchStyle.DarkVertical);
Draw(e.Graphics,rec,HatchStyle.DashedDownwardDiagonal);
Draw(e.Graphics,rec,HatchStyle.DashedHorizontal);
Draw(e.Graphics,rec,HatchStyle.DashedUpwardDiagonal);
Draw(e.Graphics,rec,HatchStyle.DashedVertical);
Draw(e.Graphics,rec,HatchStyle.DiagonalBrick);
Draw(e.Graphics,rec,HatchStyle.DiagonalCross);
Draw(e.Graphics,rec,HatchStyle.Divot);
Draw(e.Graphics,rec,HatchStyle.DottedDiamond);
Draw(e.Graphics,rec,HatchStyle.DottedGrid);
Draw(e.Graphics,rec,HatchStyle.ForwardDiagonal);
Draw(e.Graphics,rec,HatchStyle.Horizontal);
Draw(e.Graphics,rec,HatchStyle.HorizontalBrick);
Draw(e.Graphics,rec,HatchStyle.LargeCheckerBoard);
Draw(e.Graphics,rec,HatchStyle.LargeConfetti);
Draw(e.Graphics,rec,HatchStyle.LargeGrid);
Draw(e.Graphics,rec,HatchStyle.LightDownwardDiagonal);
Draw(e.Graphics,rec,HatchStyle.LightHorizontal);
Draw(e.Graphics,rec,HatchStyle.LightUpwardDiagonal);
Draw(e.Graphics,rec,HatchStyle.LightVertical);
Draw(e.Graphics,rec,HatchStyle.Max);
Draw(e.Graphics,rec,HatchStyle.Min);
Draw(e.Graphics,rec,HatchStyle.NarrowHorizontal);
Draw(e.Graphics,rec,HatchStyle.NarrowVertical);
Draw(e.Graphics,rec,HatchStyle.OutlinedDiamond);
Draw(e.Graphics,rec,HatchStyle.Percent05);
Draw(e.Graphics,rec,HatchStyle.Percent10);
Draw(e.Graphics,rec,HatchStyle.Percent20);
Draw(e.Graphics,rec,HatchStyle.Percent25);
Draw(e.Graphics,rec,HatchStyle.Percent30);
Draw(e.Graphics,rec,HatchStyle.Percent40);
Draw(e.Graphics,rec,HatchStyle.Percent50);
Draw(e.Graphics,rec,HatchStyle.Percent60);
Draw(e.Graphics,rec,HatchStyle.Percent70);
Draw(e.Graphics,rec,HatchStyle.Percent75);
Draw(e.Graphics,rec,HatchStyle.Percent80);
Draw(e.Graphics,rec,HatchStyle.Percent90);
Draw(e.Graphics,rec,HatchStyle.Plaid);
Draw(e.Graphics,rec,HatchStyle.Shingle);
Draw(e.Graphics,rec,HatchStyle.SmallCheckerBoard);
Draw(e.Graphics,rec,HatchStyle.SmallConfetti);
```

```
  Draw(e.Graphics,rec,HatchStyle.SmallGrid);
  Draw(e.Graphics,rec,HatchStyle.SolidDiamond);
  Draw(e.Graphics,rec,HatchStyle.Sphere);
  Draw(e.Graphics,rec,HatchStyle.Trellis);
  Draw(e.Graphics,rec,HatchStyle.Vertical);
  Draw(e.Graphics,rec,HatchStyle.Wave);
  Draw(e.Graphics,rec,HatchStyle.Weave);
  Draw(e.Graphics,rec,HatchStyle.WideDownwardDiagonal);
  Draw(e.Graphics,rec,HatchStyle.WideUpwardDiagonal);
  Draw(e.Graphics,rec,HatchStyle.ZigZag);
end;

// Процедура рисования текущей области со штриховкой.
procedure TWinForm.Draw(g: Graphics; var r: Rectangle;
                        aHatchStyle: HatchStyle);
begin
  if assigned(My_HatchBrush) then My_HatchBrush.Free;
  My_HatchBrush := HatchBrush.Create(aHatchStyle,
                                     Color.Black,Color.White);
  // Заполнение области штриховкой.
  g.FillRectangle(My_HatchBrush,r);
  // Рисование разделительного прямоугольника (для наглядности).
  g.DrawRectangle(Pens.Black,r);
  // Определяем координаты.
  r.X := r.X + r.Width;
  if r.x >= self.ClientSize.Width-6 then
  begin
    r.X := 0;
    r.Y := r.Y+ r.Height;
  end;
end;

end.
```

В листинге 6.4 приведен пример использования текстурной кисти. Создается кисть уже знакомым нам методом `Create`, но в качестве параметра ему уже передается текстура (файл картинки). Результат использования текстурной кисти можно наблюдать на рис. 6.5

Листинг 6.4 ▾ Пример создания и использования текстурной кисти

```
procedure TWinForm2.TWinForm2_Paint(sender: System.Object;
                                    e: System.Windows.Forms.PaintEventArgs);
var My_Brush:TextureBrush;    // Текстурная кисть.
    image_file:Image;         // Файл с картинкой.
    i,j:integer;
```

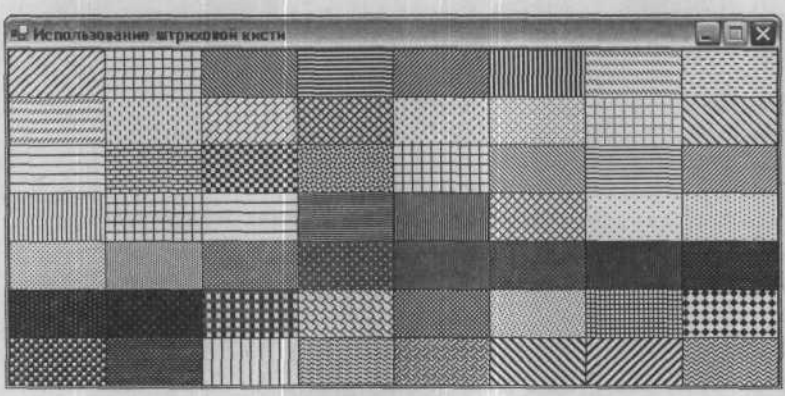

Рис. 6.4 ▼ Результат работы программы создания и использования штриховой кисти

```
begin
  // Загружаем картинку.
  image_file:=Image.FromFile('Gray_Textured.gif');
  // Создаем текстурную кисть.
  My_brush:=TextureBrush.Create(image_file);
  Width:=4*image_file.Width;
  Height:=3*image_file.Height;
  // Заполняем все пространство формы текстурой.
  for i:=0 to 3 do
    for j:=0 to 2 do
      // Рисуем текущий прямоугольник текстурной кистью.
      e.Graphics.FillRectangle(My_Brush,i*image_file.Width,
                               j*image_file.Height,
                               image_file.Width,image_file.Height);
end;
```

В листинге 6.5 приведен пример использования градиентной кисти. При создании кисти методу `Create` передаются переменная `rct`, задающая размер кисти, две константы `Color`, задающие два цвета, и параметр, определяющий градиент цвета. Градиент цвета может быть направлен по горизонтали (`LinearGradientMode.Horizontal`), вертикали (`LinearGradientMode.Vertical`), от левого верхнего угла к правому нижнему (`LinearGradientMode.BackwardDiagonal`) или от левого нижнего угла к правому верхнему (`LinearGradientMode.ForwardDiagonal`).

Листинг 6.5 ▾ Пример создания и использования градиентной кисти

```
procedure
TWinForm1.TWinForm1_Paint(
  sender: System.Object; e:
  System.Windows.Forms.PaintEventArgs);
var
   My_Brush:LinearGradientBrush;
   rct:rectangle;
begin
   rct:=rectangle.Create(0,0,30,60);
    // Создаем градиентную
    // кисть.
    My_Brush:=
  LinearGradientBrush.Create(rct,
  Color.Red,Color.Orange,
  LinearGradientMode.Horizontal);
    // Рисуем прямоугольники градиентной кистью
    // (с различным градиентом).
    e.Graphics.FillRectangle(My_Brush,30,20,270,50);
    My_Brush:=LinearGradientBrush.Create(rct,Color.Green,
                          Color.White, LinearGradientMode.Vertical);
    e.Graphics.FillRectangle(My_Brush,30,70,270,50);
    My_Brush:=LinearGradientBrush.Create(rct,Color.Yellow,Color.Blue,
                                LinearGradientMode.ForwardDiagonal);
    e.Graphics.FillRectangle(My_Brush,30,120,270,50);
  end;
```

Рис. 6.5 ▾ Результат использования текстурной кисти

Результат работы программы можно наблюдать на рис. 6.6.

Рис. 6.6 ▾ Результат использования градиентной кисти

Изучаем основные графические примитивы

Любой создаваемый нами рисунок представляет собой совокупность графических *примитивов* – точек, линий, окружностей, прямоугольников и т.п. Чтобы нарисовать подобные примитивы, используются соответствующие методы, некоторые из которых приведены в табл. 6.6.

Таблица 6.6 ▼ Методы вычерчивания графических примитивов

Метод	Действие
`DrawLine(Pen,x1,y1,x2,y2)`	Рисует линию. Параметр `Pen` определяет цвет, толщину и стиль линии. Параметры `x1`, `y1`, `x2`, `y2` определяют координаты начальной и конечной точки линии
`DrawRectangle(Pen,x1,y1,x2,y2)`	Рисует контур прямоугольника. Параметр `Pen` также задает карандаш, а параметры `x1`, `y1`, `x2`, `y2` – координаты находящихся на одной диагонали углов прямоугольника
`DrawEllipse(Pen,x1,y1,x2,y2)`	Рисует контур эллипса. Параметр `Pen` задает карандаш, параметры `x1`, `y1`, `x2`, `y2` – координаты диагональных углов области, внутри которой вычерчивается эллипс
`DrawArc(Pen,x1,y1,x2,y2,angle_1,angle_2)`	Рисует контур дуги. Параметр `Pen` задает карандаш, параметры `x1`, `y1`, `x2`, `y2` – координаты диагональных углов области, внутри которой вычерчивается эллипс, параметры `angle_1` – начальный угол, `angle_2` – длину дуги в градусах
`DrawPie(Pen,x1,y1,x2,y2,angle1,angle_2)`	Рисует контур сектора. Параметр `Pen` задает карандаш, параметры `x1`, `y1`, `x2`, `y2` – координаты диагональных углов области, внутри которой вычерчивается эллипс, параметры `angle_1` – начальный угол, `angle_2` – длину дуги окружности, формирующей сектор, в градусах
`DrawPolygon(Pen,P_Array)`	Рисует контур многоугольника. Параметр `Pen` задает карандаш, а параметр `P_Array` типа `Point` – массив точек (`x`, `y`) – координаты углов многоугольника
`DrawString(st,Font,Brush,x,y)`	Выводит строку текста. Параметр `Font` определяет шрифт текста, `Brush` – цвет символов, `x` и `y` – координаты точки, от которой будет выводиться текст
`DrawImage(Image,x,y)`	Выводит изображение. Параметр `Image` определяет выводимое изображение, `x` и `y` – координаты точки, от которой будет осуществляться вывод

Таблица 6.6 ▾ Методы вычерчивания графических примитивов (окончание)

Метод	Действие
`FillRectangle(Brush,x1,y1,x2,y2)`	Рисует закрашенный прямоугольник. Параметры идентичны рассмотренным ранее методам, разница состоит в том, что вместо карандаша используется кисть
`FillEllipse(Brush,x1,y1,x2,y2)`	Рисует закрашенный эллипс
`FillPie(Brush,x1,y1,x2,y2,angle_1,angle_2)`	Рисует закрашенный сектор
`FillPolygon(Brush,x1,y1,x2,y2)`	Рисует закрашенный многоугольник

Необходимо заметить, что один и тот же примитив можно нарисовать при помощи разных (хотя и имеющих одинаковые имена) методов. Например, прямоугольник может быть нарисован методом `DrawRectangle`, которому в качестве параметров передаются координаты точек, находящихся на одной диагонали (см. табл. 6.6):

```
e.Graphics.DrawRectangle(Pens.Black,x1,x2,y1,y2);
```

Того же результата можно добиться, если в качестве параметра этому методу передать структуру типа `Rectangle`, параметры которой задают положение левого верхнего угла и размер прямоугольника:

```
e.Graphics.DrawRectangle(Pens.Black,My_rect);
```

Существование нескольких методов, по сути выполняющих одну и ту же задачу, зачастую предоставляет программисту возможность выбора наиболее удобного метода.

В качестве параметров методов вычерчивания графических примитивов используются структуры `Point` и `Rectangle`. Структура `Point`, которую образуют поля `x` и `y`, служит для передачи в метод координат точек плоскости (графической поверхности), структура `Rectangle` (ее образуют поля `x`, `y`, `Width` и `Height`) – для передачи в метод информации о положении и размере прямоугольной области. В качестве примера использования структур `Point` и `Rectangle` приведу фрагмент программы (листинг 6.6).

Листинг 6.6 ▾ Пример использования структур `Point` и `Rectangle`

```
procedure TWinForm.TWinForm_Paint(sender: System.Object;
                                  e: System.Windows.Forms.PaintEventArgs);
var
  p1,p2:Point;          // Точки начала и конца линии.
  My_rect:Rectangle;    // Прямоугольная область.
begin
  p1.X:=100;p1.Y:=10; // Координаты начальной точки линии.
```

```
  p2.X:=200;p2.Y:=30; // Координаты конечной точки линии.
  e.Graphics.DrawLine(Pens.Blue,p1,p2);    // Рисуем линию.
  // Координаты левого верхнего угла прямоугольника.
  My_rect.X:=20;
  My_rect.Y:=20;
  My_rect.Width:=60;              // Длина прямоугольника.
  My_rect.Height:=40;             // Высота прямоугольника.
  e.Graphics.DrawRectangle(Pens.Red,My_rect); // Рисуем
                                              // прямоугольник.
end;
```

Далее подробно разберем методы вычерчивания графических примитивов.

Линия

Для рисования прямой линии используется метод `DrawLine` (`Line` в переводе с англ. – линия). В качестве параметров метода используется карандаш, а также координаты точек начала и конца линии:

```
DrawLine(My_Pen,x1,y1,x2,y2);
```

или

```
DrawLine(My_pen,_p1,_p2);
```

В последнем случае используются структуры `_p1` и `_p2` типа `Point`.

Например, мы хотим нарисовать линию толщиной в один пиксель из точки (0,0) в точку (100,100). Тогда нарисовать нашу линию можно инструкцией:

```
e.Graphics.Drawline(Pens.Black,0,0,100,100);
```

Также можно воспользоваться вторым методом:

```
_p1.x:=0;_p2.x:=100;
_p1.y:=0;_p2.y:=100;
e.Graphics.Drawline(Pens.Black,_p1,_p2);
```

Прямоугольник

Прямоугольник можно начертить методом `DrawRectangle` (`Rectangle` в переводе с англ. – прямоугольник). В качестве параметров этого метода указывается карандаш и координаты двух углов, расположенных на одной диагонали:

```
DrawRectangle(My_Pen,x1,y1,x2,y2);
```

Того же результата можно добиться, использовав структуру `Rectangle`:

```
DrawRectangle(My_pen,My_Rect);
```

Поля x и y структуры Rectangle содержат координаты левого верхнего угла прямоугольника, а поля Width и Height – соответственно его длину и высоту. Внешний же вид прямоугольника задается параметрами карандаша, в качестве которого может быть использован стандартный карандаш или же карандаш, созданный программистом.

Для рисования закрашенного прямоугольника следует воспользоваться методом FillRectangle. Параметры этого методы аналогичны предыдущему, основное отличие состоит в том, что в данном случае уже используется инструмент *кисть*, а не карандаш. Соответственно возможно использование как стандартной, так и других типов кисти, рассмотренных нами ранее. Приведенный ниже в листинге 6.7 фрагмент исходного кода демонстрирует использование рассмотренных выше методов DrawRectangle и FillRectangle.

Листинг 6.7 ▾ Использование методов DrawRectangle и FillRectangle

```
procedure TWinForm.TWinForm_Paint(sender: System.Object;
                          e: System.Windows.Forms.PaintEventArgs);
var
  My_rect:rectangle; // Прямоугольная область.
begin
  My_rect:=Rectangle.Create(0,0,100,80); // Создаем прямоугольную
                                         // область.
  e.Graphics.FillRectangle(Brushes.Red,My_rect); // Выводим
                                                 // закрашенный
                                                 // прямоугольник.
  e.Graphics.DrawRectangle(Pens.Black,My_Rect);  // Выводим
                                                 // прямоугольный
                                                 // контур.
end;
```

Многоугольник

Многоугольник можно нарисовать методом DrawPolygon. При этом вычерчивается контур многоугольника. Инструкция рисования многоугольника выглядит следующим образом:

```
DrawPolygon(My_Pen,M_points);
```

Параметр M_points представляет собой массив из элементов типа Point. Эти элементы являются узловыми вершинами (так же, как и в случае рисования ломаной линии). Вершина, являющаяся последней в массиве, соединяется с первой вершиной.

Если необходимо нарисовать закрашенный многоугольник, то следует воспользоваться методом `FillPolygon`. Инструкция рисования закрашенного многоугольника выглядит аналогично:

```
FillPolygon(My_Brush,M_points);
```

Приведенный ниже в листинге 6.8 фрагмент исходного кода демонстрирует использование рассмотренных выше методов `DrawPolygon` и `FillPolygon`.

Листинг 6.8 ▼ Использование методов `DrawPolygon` и `FillPolygon`

```
procedure TWinForm.TWinForm_Paint(sender: System.Object;
                                  e: System.Windows.Forms.PaintEventArgs);
var pts:array [1..4] of Point;
    My_pen:Pen;
begin
  // Задание узловых точек многоугольника.
  pts[1].X:=10; pts[1].Y:=10;
  pts[2].X:=50; pts[2].Y:=40;
  pts[3].X:=40; pts[3].Y:=40;
  pts[4].X:=30; pts[4].Y:=60;
  // Рисуем закрашенный многоугольник.
  e.Graphics.FillPolygon(Brushes.Red,pts);
  // Рисуем его контур.
  My_pen:=Pen.Create(Color.Blue,2);
  e.Graphics.DrawPolygon(My_pen,pts);
end;
```

Окружность и эллипс

Эллипс можно нарисовать методом `DrawEllipse`. Этот же метод рисует и окружность, которая является частным случаем эллипса. В качестве параметров метода указываются карандаш и прямоугольная область, ограничивающая фигуру (рис. 6.7):

```
DrawEllipse(My_Pen,x1,y1,x2,y2);
```

или

```
DrawEllipse(My_Pen,My_Rect);
```

В первом случае область задается координатами углов, расположенных на одной диагонали (см. рис. 6.2, слева), во втором случае для задания прямоугольной области используется структура типа `Rectangle` (см. рис. 6.7, справа).

По аналогии с приведенными выше примерами можно нарисовать закрашенный эллипс. Для этого следует воспользоваться методом `FillEllipse`:

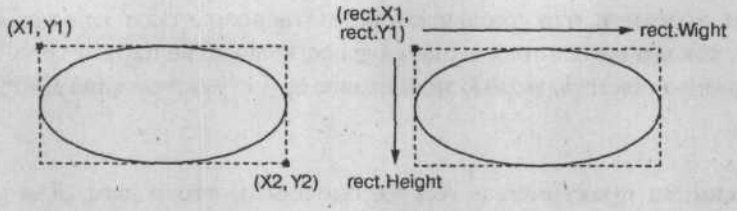

Рис. 6.7 ▾ Различные способы рисования эллипса

```
FillEllipse(My_Brush,x1,y1,x2,y2);
```

или

```
FillEllipse(My_Brush,My_Rect);
```

Дуга

Дуга представляет собой часть эллипса. Поэтому в методе `DrawArc`, который позволяет нарисовать дугу, в качестве параметров необходимо дополнительно указать два угла, определяющих начальную и конечную точку дуги. Инструкция вызова метода `DrawArc` выглядит следующим образом:

```
DrawArc(My_pen,x1,y1,x2,y2,Angle1,Angle2);
```

Параметры `x1`, `y1`, `x2`, `y2` определяют эллипс, частью которого является дуга. `Angle1` задает начальную точку дуги, которая является пересечением эллипса и прямой, проведенной из его центра и образующей угол `Angle1` с его горизонтальной осью. Параметр `Angle2` задает длину дуги в градусах (рис. 6.8).

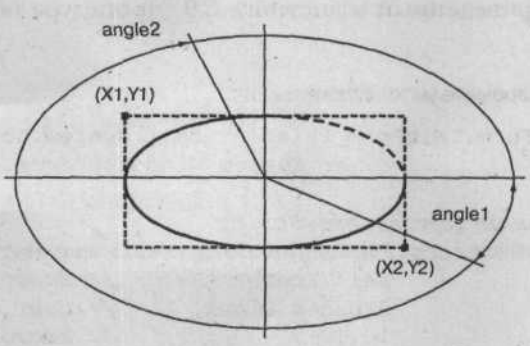

Рис. 6.8 ▾ Способ задания дуги

Следует отметить, что угловые координаты возрастают *по часовой стрелке*.

Так же, как и в описанных выше случаях, в качестве параметров, задающих прямоугольную область, можно использовать и структуру типа `Rectangle`.

Сектор

Сектор задается практически тем же способом, что и дуга. Для рисования сектора необходимо использовать метод `DrawPie`:

```
DrawPie(My_pen,x1,y1,x2,y2,Angle1,Angle2);
```

Ко всему вышесказанному остается добавить, что для задания прямоугольной области также можно воспользоваться структурой типа `Rectangle`, а для рисования закрашенного сектора – методом `FillPie`. Параметры метода `FillPie` аналогичны, за исключением того, что в качестве основы для рисования необходимо использовать кисть, а не карандаш.

Вставка текста

Текст на графическую поверхность можно вставить с помощью метода `DrawString`:

```
DrawString(My_string,_Font,_Brush,x,y);
```

Параметр `My_string` содержит собственно строку, которую необходимо отобразить. С помощью параметра `_Font` задается шрифт текста, а с помощью параметра `_Brush` – цвет текста. Место вывода текста на экране определяется параметрами `x` и `y`.

В качестве примера использования метода `DrawString` можно привести фрагмент кода, приведенный в листинге 6.9 (процедура обработки события `Paint`).

Листинг 6.9 ▼ Использование метода `DrawString`

```
procedure TWinForm.TWinForm_Paint(sender: System.Object;
                                  e: System.Windows.Forms.PaintEventArgs);
begin
  // Использование метода DrawString.
  e.Graphics.DrawString('Вывод надписи с помощью метода DrawString',
                        self.Font,          // Шрифт.
                        Brushes.Black,      // Цвет символов.
                        10,10);             // Место вывода.
end;
```

В приведенном примере для вывода текста используется шрифт, заданный для формы приложения (свойство Font формы). Идентификатор Self обозначает объект, членом которого является данный метод и используется для доступа к объекту (к свойствам объекта).

Если необходимо вывести текст шрифтом, отличным от шрифта формы, то он создается самим программистом. Для этого следует создать объект типа Font, принадлежащий пространству имен System.Drawing:

```
My_font:=System.Drawing.Font.Create(System_Font,Size,FontStyle);
```

Параметр строкового типа System_Font задает шрифт, на основе которого задается новый. В качестве значения этого параметра можно использовать названия шрифтов, зарегистрированных в системе, – Arial, Courier, Times New Roman и т.д. Параметр Size задает размер шрифта, а FontStyle – стиль символов шрифта. Стиль может быть полужирным (FontSyle.Bold), курсивом (FontSyle.Italic), подчеркнутым (FontSyle.UnderLine). Если не указывать параметр FontSyle, то текст будет выведен обычным начертанием.

В листинге 6.10 приведен фрагмент кода (процедура обработки события Paint), поясняющий использование дополнительных возможностей метода DrawString. Результат выполнения этого кода приведен на рис. 6.9.

Листинг 6.10 ▾ Еще один пример использования метода DrawString

```
procedure TWinForm.TWinForm_Paint(sender: System.Object;
                                  e: System.Windows.Forms.PaintEventArgs);
var f_normal:System.Drawing.Font;    // Обычный шрифт.
    f_bold:System.Drawing.Font;      // Полужирный шрифт.
    f_italic:System.Drawing.Font;    // Курсив.
    f_underline:System.Drawing.Font;// Подчеркнутый.
begin
  // Создание шрифтов.
  f_normal:=System.Drawing.Font.Create('Courier New',12);
  f_bold:=System.Drawing.Font.Create('Courier New',12,
                                     FontStyle.Bold);
  f_italic:=System.Drawing.Font.Create('Courier New',12,
                                       FontStyle.Italic);
  f_underline:=System.Drawing.Font.Create('Courier New',12,
                                         FontStyle.Underline);
  // Используем шрифты для вывода надписей.
  // Обычный шрифт.
```

```
  e.Graphics.DrawString('Вывод надписи с помощью метода DrawString',
                        F_Normal,          // Шрифт.
                        Brushes.Black,     // Цвет символов.
                        10,10);            // Место вывода.
  // Полужирный шрифт.
  e.Graphics.DrawString('Вывод надписи с помощью метода DrawString',
                        F_Bold,            // Шрифт.
                        Brushes.Black,     // Цвет символов.
                        10,40);            // Место вывода.
  // Курсив.
  e.Graphics.DrawString('Вывод надписи с помощью метода DrawString',
                        F_Italic,          // Шрифт.
                        Brushes.Black,     // Цвет символов.
                        10,70);            // Место вывода.
  // Подчеркнутый.
  e.Graphics.DrawString('Вывод надписи с помощью метода DrawString',
                        F_Underline,       // Шрифт.
                        Brushes.Black,     // Цвет символов.
                        10,100);           // Место вывода.
end;
```

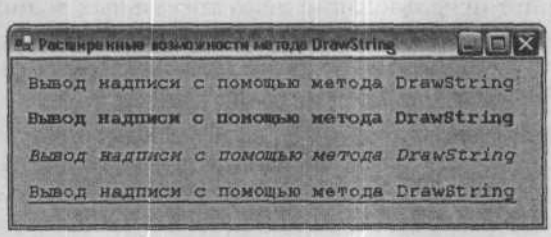

Рис. 6.9 ▼ Использование дополнительных возможностей метода `DrawString`

Знакомимся с основными приемами мультипликации

Картинку, которая находится в графическом файле, проще всего отобразить с помощью компонента `PictureBox`. Подробное описание компонента, а также примеры его использования можно найти в главе 5. Поэтому мы не будем останавливаться сейчас на основах работы с файлами, а перейдем сразу к возможностям создания эффекта мультипликации.

Под мультипликацией обычно понимается изменяющийся рисунок (картинка). Эффект мультипликации достигается за счет перемещения объектов – элементов рисунка.

Существует два подхода к реализации данного эффекта.

В первом случае предполагается создание кадров в процессе работы программы. При таком подходе кадры создаются путем вывода изображений (битовых образов) в определенное место экрана, как правило, имеющее фоновый рисунок. Такой подход широко используется разработчиками компьютерных игр.

Второй подход заключается в том, что отображаемые кадры подготавливаются заранее и в нужный момент последовательно отображаются. Этот подход используют в основном создатели мультфильмов. В среде разработки Borland Delphi 2005 воспроизведение классической анимации, выполненной в виде gif-файла, можно осуществить с помощью компонента `PictureBox`.

Использование битовых образов

Для формирования изображений используют *битовые образы*. Битовый образ – это картинка, которая хранится в памяти компьютера. Битовый образ создается путем загрузки из файла графического формата (bmp, jpg и т.д.), ресурса или же путем копирования из другого графического объекта.

Создать битовый образ можно с помощью метода `Create`. В качестве параметра этому методу необходимо передать, например, имя файла, в котором хранится изображение. Следующий фрагмент исходного кода демонстрирует создание битового образа:

```
var _image:Bitmap;
…
begin
…
_image:=Bitmap.Create('source.bmp');
…
end;
```

Таким образом, битовый образ сформирован. Что же дальше? Дальше все просто – картинка хранится в памяти компьютера и ее можно вывести на поверхность формы методом `DrawImage`. В качестве параметров нужно указать битовый образ, который мы собираемся отображать, и координаты точки

поверхности, с которой будет осуществляться вывод. Следующий оператор позволяет вывести битовый образ в левом верхнем углу формы:

```
e.Graphics.DrawImage(_image,0,0);
```

Вместо координат точек возможно использование структуры типа `Point`.

Кроме того, для битового образа также можно задать *прозрачный* цвет. Точки картинки, цвет которых совпадает с прозрачным, отображаться не будут. Например, оператор

```
_image.MakeTransparent(Color.White);
```

указывает, что для битового образа `_image` прозрачным является белый цвет. В качестве параметра метода `MakeTransparent` можно указать цвет какой-либо точки битового образа:

```
_image.MakeTransparent(_image.GetPixel(1,1));
```

Данная инструкция задает прозрачным цвет точки с координатами (1,1). В инструкции вызова метода `MakeTransparent` цвет можно и не указывать. Тогда прозрачным будет цвет точки, находящейся в левом нижнем углу битового образа.

Следующая программа (листинг 6.11) демонстрирует создание и отображение битовых образов.

Листинг 6.11 ▼ Работа с битовыми образами

```
// Обработка события Load.
procedure TWinForm.TWinForm_Load(sender: System.Object;
                                 e: System.EventArgs);
begin
  // Загружаем фон.
  sky:=Bitmap.Create('sky.bmp');
  // Изменяем размер окна под размер фоновой картинки.
  Self.Width:=sky.Width;
  Self.Height:=sky.Height;
  // Загружаем изображение парашютиста.
  parashut:=Bitmap.Create('parashut.bmp');
end;
// Обработка события Paint.
procedure TWinForm.TWinForm_Paint(sender: System.Object;
                          e: System.Windows.Forms.PaintEventArgs);
var
  p:Point; // Координаты вывода битового образа.
```

```
begin
  // Отображаем фон (небо).
  e.Graphics.DrawImage(sky,0,0);
  // Отображаем левого парашютиста.
  p.X :=100; p.Y := 50;
  e.Graphics.DrawImage(parashut,p);
  // Отображаем правого парашютиста.
  p.X :=200; p.Y:=80;
  // Устанавливаем 'прозрачный' цвет
  // (цвет левой нижней точки).
  parashut.MakeTransparent;
  e.Graphics.DrawImage(parashut,p);
end;
```

В результате окно нашей программы будет выглядеть так, как показано на рис. 6.10.

Рис. 6.10 ▾ Использование битовых образов и установки прозрачного цвета

После того как мы научились отображать битовые образы, самое время перейти к мультипликации. Для этого нам потребуется последовательно отображать битовые образы в разных местах экрана, тем самым создавая иллюзию движения.

Для того чтобы у наблюдателя складывалось впечатление о движении объекта, необходимо периодически перерисовывать изображение объекта со смещением относительно предыдущего положения. Следовательно, необходимо периодически фиксировать событие, процедура обработки которого и осуществляла бы

перерисовку. Такое событие можно получить, например, использованием компонента Timer.

Приведенный в листинге 6.12 исходный код демонстрирует данный прием мультипликации.

Листинг 6.12 ▼ Мультипликация с помощью битовых образов

```
unit WinForm;

interface

uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data, System.Resources;

type
  TWinForm = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    components: System.ComponentModel.IContainer;
    Timer1: System.Windows.Forms.Timer;
    procedure InitializeComponent;
    procedure TWinForm_Load(sender: System.Object;
                            e: System.EventArgs);
    procedure Timer1_Tick(sender: System.Object;
                          e: System.EventArgs);
  {$ENDREGION}
  strict protected
    procedure Dispose(Disposing: Boolean); override;
  public
    constructor Create;
  end;

implementation

uses
  System.Globalization;
const dy=1;          // Шаг смещения парашютиста.
var
  sky,parashut:System.Drawing.Bitmap; // Битовые образы.
  rect:rectangle;  // Копируемая прямоугольная область.
  d:graphics;      // Графическая поверхность для рисования.

{$REGION 'Windows Form Designer generated code'}
procedure TWinForm.Dispose(Disposing: Boolean);
begin
  if Disposing then
```

```
  begin
    if Components <> nil then
      Components.Dispose();
  end;

  inherited Dispose(Disposing);
end;
constructor TWinForm.Create;
begin
  inherited Create;
  InitializeComponent;
end;
// Процедура обработки события таймера.
procedure TWinForm.Timer1_Tick(sender: System.Object;
                               e: System.EventArgs);
begin
  // Стираем парашютиста с исходной позиции.
  d.DrawImage(sky,rect,rect,GraphicsUnit.Pixel);
  // Прибавляем к координате значение шага.
  rect.Y:=rect.Y+dy;
  // Если вертикальная координата >160, то заканчиваем движение.
  if rect.Y>160 then
  begin
    Timer1.Enabled:=False;
  end;
  // Рисуем парашютиста на новом месте.
  d.DrawImage(Parashut,rect.X,rect.Y);
  // Обновляем содержимое формы
  // (точнее - прямоугольного участка формы rect).
  Self.Invalidate(Rect);
end;

// Обработка события Load.
procedure TWinForm.TWinForm_Load(sender: System.Object;
                                 e: System.EventArgs);
begin
  // Загружаем фон в объект sky.
  sky:=Bitmap.Create('sky.bmp');
  // Изменяем размер окна под размер фоновой картинки.
  Self.Width:=sky.Width;
  Self.Height:=sky.Height;
  // Загружеаем фоновый рисунок из файла.
  Self.BackgroundImage:=Image.FromFile('sky.bmp');
  // Загружаем изображение парашютиста.
  parashut:=Bitmap.Create('parashut.bmp');
```

```
  parashut.MakeTransparent;
  // Вводим координаты положения парашютиста
  rect.X:=100;
  rect.Y:=10;
  // и размеры копируемого фрагмента.
  rect.Width:=Parashut.Width;
  rect.Height:=Parashut.Height;
  d:=Graphics.FromImage(self.BackgroundImage);
  // Запускаем таймер.
  Timer1.Interval:=60;
  Timer1.Enabled:=True;
end;

end.
```

Программа работает следующим образом. Сначала объект выводится на созданную нами графическую поверхность `d`, через некоторое время стирается и выводится снова, но уже на некотором расстоянии `dy` от первоначального положения. Таким образом, парашютист «движется» вертикально (рис. 6.11).

Необходимо отметить, что перерисовку формы можно производить разными способами. Можно применить к форме метод `Refresh` – это наиболее простой способ. Но данный метод перерисовывает *все* окно, а в этом нет необходимости – перерисовывать требуется только часть окна, занимаемую парашютистом, – прямоугольную область `rect`. Поэтому в программе применяется другой метод – `Invalidate`. Этот метод также выполняет перерисовку, но ему в качестве параметра перерисовки указывается область, которую надо перерисовать (`rect`).

Рис. 6.11 ▼ Движение битового образа

Использование GIF-анимации

Использование этого приема построено на особенности графического формата GIF. Этот формат позволяет хранить в одном файле несколько изображений, поэтому его часто используют для хранения кадров анимации.

Отображается файл данного формата точно так же, как и все остальные файлы изображений, – с помощью компонента PictureBox. Если нет необходимости управлять процессом смены и отображения кадров, то загрузки файла в данный компонент достаточно. При этом управлять процессом смены и отображения кадров будут параметры анимации, заданные во время создания GIF-файла.

Если есть необходимость управлять процессом отображения анимации, то нужно использовать объект ImageAnimator. Использование этого объекта поясним на примере (листинг 6.13).

Листинг 6.13 ▾ Использование GIF-анимации

```
unit WinForm;

interface

uses
  System.Drawing, System.Collections, System.ComponentModel,
  System.Windows.Forms, System.Data, System.Resources;

type
  TWinForm = class(System.Windows.Forms.Form)
  {$REGION 'Designer Managed Code'}
  strict private
    Components: System.ComponentModel.Container;
    PictureBox1: System.Windows.Forms.PictureBox;
    Button1: System.Windows.Forms.Button;
    Label1: System.Windows.Forms.Label;
    procedure InitializeComponent;
  {$ENDREGION}
    procedure OnFrameChanged(sender: System.Object;
                             e: System.EventArgs);
    procedure Button1_Click(sender: System.Object;
                            e: System.EventArgs);
    procedure PictureBox1_Paint1(sender: System.Object;
      e: System.Windows.Forms.PaintEventArgs);
  strict protected
    procedure Dispose(Disposing: Boolean); override;
  public
```

```
    constructor Create;
  end;

implementation

uses
  System.Globalization;

var
  img_animator:ImageAnimator;      // Объект ImageAnimator.
  gif_img:System.Drawing.Bitmap; // Битовый образ gif-файла.
  ev_handler:EventHandler;         // Указатель на процедуру
                                   // обработки события
                                   // объекта ImageAnimator.
  p: Point;                        // Координаты верхней левой
                                   // точки, с которой
                                   // будет происходить вывод
                                   // анимации.

{$REGION 'Windows Form Designer generated code'}
procedure TWinForm.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
  inherited Dispose(Disposing);
end;
procedure TWinForm.OnFrameChanged(sender: System.Object;
                                  e: System.EventArgs);
begin
  // Инициируем рисование кадра.
  PictureBox1.Invalidate;
end;

constructor TWinForm.Create;
begin
  inherited Create;
  InitializeComponent;
  // Создаем битовый образ gif-файла.
  gif_img:=System.Drawing.bitmap.Create('abc.gif');
  // Инициируем указатель на процедуру обработки события
  // OnFrameChanged, генерируемого объектом ImageAnimator.
  ev_handler:=OnFrameChanged;
  p:= Point.Create(1,1);
```

```
end;
procedure TWinForm.PictureBox1_Paint1(sender: System.Object;
                          e: System.Windows.Forms.PaintEventArgs);
begin
  // Отображаем текущий кадр анимации.
  e.Graphics.DrawImage(gif_img,p);
  // Подготавливаем очередной кадр анимации.
  img_animator.UpdateFrames;
end;

// Нажимаем на кнопку 'Анимация'/'Стоп'.
procedure TWinForm.Button1_Click(sender: System.Object;
                                 e: System.EventArgs);
begin
  if Button1.Text  = 'Стоп' then
    begin
      // Останавливаем воспроизведение.
      img_animator.StopAnimate(gif_img,ev_handler);
      Button1.Text := 'Анимация';
    end
  else
    begin
      // Начинаем воспроизведение.
      img_animator.Animate(gif_img,ev_handler);
      Button1.Text := 'Стоп';
    end;
end;

end.
```

Данная программа, форма диалогового окна которой приведено на рисунке 6.12, воспроизводит анимацию с помощью компонента `PictureBox`. Процедура обработки события `Create` для формы загружает анимацию и инициализирует указатель на процедуру, обеспечивающую отображение очередного кадра анимации. Процесс воспроизведения анимации активизирует процедура обработки события `Click` на командной кнопке `Button1` путем вызова метода `Animate`. В качестве параметров метода указываются битовый образ и ссылка на процедуру, обеспечивающую отображение кадра анимации.

Рис. 6.12 ▾ Форма окна программы, демонстрирующей пример использования объекта `ImageAnimator`

В тот момент, когда необходимо вывести очередной кадр анимации, объект ImageAnimate вызывает процедуру, ссылка на которую была указана в методе Animate (в данном случае – на процедуру FrameChanged). Процедура FrameChanged в свою очередь с помощью метода Invalidate инициализирует процедуру обработки события Paint, которая обеспечивает вывод текущего кадра анимации и подготовку следующего кадра. Останавливается процесс анимации с помощью метода StopAnimate.

Глава 7 Изучаем основные компоненты при программировании для Win32

В этой главе приведено описание основных компонентов, которые используются для создания программ на платформе Win32. Как и в главе 5, здесь будут приводиться примеры программ, в которых используется сразу несколько компонентов. При этом основное внимание уделяется компоненту, название которого вынесено в заголовок раздела. В таблицах приводятся основные свойства, отражающие специфику компонента и представляющие для начинающего программиста наибольший интерес. Подробную информацию о компонентах можно найти в справочной системе.

Основные компоненты располагаются на вкладках **Standard** и **Additional** окна **Tool Palette**. Некоторые полезные компоненты располагаются на других вкладках – в этом случае я буду указывать дополнительно их местоположение. Часть компонентов нам уже знакома по первому проекту – они кратко будут рассмотрены вначале, на незнакомых компонентах мы остановимся более подробно.

Прежде чем мы перейдем к рассмотрению основных компонентов, обратите внимание на табл. 7.1. В этой таблице перечислены все рассмотренные в главе 5 компоненты (.NET), а также приведены их аналоги (Win32), выполняющие по сути те же действия, но имеющие отличия в реализации.

Таблица 7.1 ▾ Компоненты .NET и их аналоги в Win32

Компонент .NET	Компонент Win32	Описание
`Label`	`TLabel`	Компонент отображения текстовой информации
`TextBox`	`TEdit`	Компонент для ввода информации (текстовой, числовой и т.д.)

Таблица 7.1 ▼ Компоненты .NET и их аналоги в Win32 (окончание)

Компонент .NET	Компонент Win32	Описание
Button	TButton	Командная кнопка
ImageList	TImageList	Компонент-контейнер для хранения коллекции картинок
ToolTip	-	Вспомогательный компонент, всплывающая подсказка
Panel	TPanel	Компонент-контейнер для объединения компонентов
CheckBox	TCheckBox	Компонент-переключатель, имеющий 2 или 3 состояния
RadioButton	TRadioButton, TRadioGroup	Компонент-переключатель, имеющий 2 состояния
GroupBox	TGroupBox	Компонент-контейнер для объединения других компонентов по функциональному признаку
ComboBox	TComboBox	Компонент для ввода информации. Представляет собой сочетания поля для ввода и выпадающего списка
ListBox	TListBox	Компонент-список, предоставляющий возможность выбора своих элементов
CheckedListBox	TCheckListBox	Компонент-список, состоящий из переключателей CheckBox
PictureBox	TImage	Компонент для отображения картинок (графических файлов)
NumericUpDown	TUpDown	Компонент для ввода числовой информации
StatusBar	TStatusBar	Компонент для отображения служебной информации
Timer	TTimer	Компонент, генерирующий циклические события с определенным интервалом
ToolBar	TToolBar	Компонент, представляющий собой панель для размещения на ней командных кнопок
ProgressBar	TProgressBar	Компонент отображения протекания длительных процессов
MainMenu	TMainMenu	Компонент, представляющий собой главное меню приложения
ContextMenu	TPopupMenu	Компонент, представляющий собой контекстное (всплывающее) меню
OpenFileDialog	TOpenDialog	Компонент, обеспечивающий вывод на экран стандартного диалогового окна открытия файла
SaveFileDialog	TSaveDialog	Компонент, обеспечивающий вывод на экран стандартного диалогового окна сохранения файла

Как видите, практически все компоненты имеют свои аналоги. Исключение составляет компонент ToolTip. В Win32 всплывающая подсказка реализуется без дополнительного компонента. Каждый визуальный компонент Win32 имеет два свойства: Hint и ShowHint. Первое свойство определяет текст подсказки, который будет выведен при наведении мыши на компонент, второе – надобность в отображении текста подсказки для данного компонента. Если значение свойства ShowHint установлено в True, то подсказка будет отображаться, в

противном случае (даже если свойство Hint не пустое) отображение подсказки не произойдет.

Напомню, что в данной книге мы рассматриваем далеко не все компоненты. Поскольку назначение этой книги состоит в том, чтобы *научить* использовать компоненты, то здесь приведены только самые типовые (основные) из них. После прочтения этой книги вам будет вполне по силам получить самостоятельно информацию по остальным компонентам Borland Delphi 2005.

Итак, переходим к рассмотрению компонентов. Начинаем рассмотрение с уже известных нам по первым проектам компонентов – TLabel, TEdit и TButton.

Компонент TLabel

С компонентом TLabel мы успели познакомиться при создании нашего первого проекта. Как вы уже знаете, этот компонент предназначен для отображения текстовой информации. Текст, который будет отображен, можно задавать как на этапе разработки формы, так и в процессе выполнения программы, присвоив значение свойству Caption. Основные свойства компонента приведены в табл. 7.2.

Таблица 7.2 ▾ Основные свойства компонента TLabel

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Caption	Текст, отображаемый в поле компонента
Font	Шрифт, который используется для отображения текста
Color	Цвет фона поля компонента
Left	Расстояние от левой границы формы до левой границы компонента
Top	Расстояние от верхней границы формы до верхней границы компонента
Width	Ширина поля компонента
Height	Высота поля компонента
AutoSize	Свойство, определяющее, будет ли размер компонента зависеть от текста в поле компонента. В случае AutoSize=True размер компонента будет ограничен текстом в поле компонента. В противном случае размер компонента определяется значениями свойств Width и Height
Transparent	Признак необходимости задания прозрачного цвета для поля компонента. Если Transparent=True, то цвет компонента определяется цветом поверхности, на которой он находится. В противном случае цвет определяется значением свойства Color

Таблица 7.2 ▼ Основные свойства компонента TLabel (окончание)

Свойство	Комментарий
Align	Определяет границу, к которой будет «прижат» компонент. Он может быть прижат к верхнему краю (alTop), к нижнему краю (alBottom), левому краю (alLeft), правому краю (alRight), быть «растянутым» на всю форму (alClient). Также свойство может иметь значение None – положение и размер компонента определяется свойствами Top, Left, Width и Height
Alignment	Способ выравнивания текста в поле компонента. Текст может быть прижат к левому краю (taLeftJustify), к правому краю (taRightJustify) или находиться посередине (taCenter)

Чтобы в поле компонента TLabel вывести числовое значение, его необходимо при помощи функций IntToStr или FloatToStr преобразовать в строку. Первая функция позволяет переводить целые числа в строку, вторая – вещественные.

Цвет поля компонента (Color) можно задать, указав название цвета (clGreen, clRed, clBlue и т.п.) или элемент цветовой схемы операционной системы (например, clBackGround, clBtnFace). Разница состоит в том, что во втором случае цветовая схема вашего приложения будет привязана к цветовой схеме операционной системы и будет автоматически меняться при каждой ее смене. По умолчанию используется как раз второй вариант. Кстати, с помощью функции RGB можно задать *любой* цвет. Эта функция «переводит» цвет, представленный тремя составляющими (R – Red, красный; G – Green, зеленый; B – Blue, голубой) в номер цвета, понятный Delphi. Цвет поля компонента также может быть и «прозрачным». Для этого необходимо свойство Transparent установить в True.

Теперь рассмотрим небольшую программу, демонстрирующую нам основные свойства этого компонента в действии. На рис. 7.1 приведен внешний вид окна программы, а в листинге 7.1 – процедуры обработки события Click на соответствующих кнопках.

Листинг 7.1 ▼ Текст процедур обработки событий Click на кнопках

```
// Текст выравнивается по левому краю.
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
  Label1.Alignment:=taLeftJustify;
end;

// Текст выравнивается по центру.
procedure TForm1.RadioButton2Click(Sender: TObject);
begin
  label1.Alignment:=taCenter;
end;
```

```
// Текст выравнивается по правому краю.
procedure TForm1.RadioButton3Click(Sender: TObject);
begin
  Label1.Alignment:=taRightJustify;
end;

// Компонент прижимается к верхней границе формы.
procedure TForm1.RadioButton4Click(Sender: TObject);
begin
  Label1.Align:=alTop;
end;

// Компонент прижимается к нижней границе формы.
procedure TForm1.RadioButton5Click(Sender: TObject);
begin
  Label1.Align:=alBottom;
end;
// Восстановление исходных значений свойств компонента.
procedure TForm1.RadioButton6Click(Sender: TObject);
begin
  Label1.Align:=alNone;
  Label1.Left:=16;
  Label1.Top:=16;
  Label1.Width:=401;
  Label1.Height:=57;
end;

// Установка/сброс отображения всплывающей подсказки.
procedure TForm1.CheckBox1Click(Sender: TObject);
```

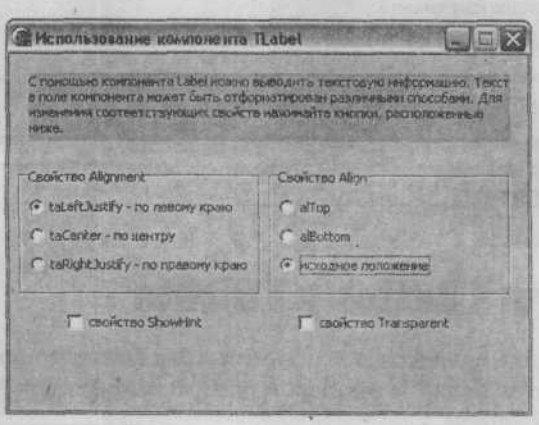

Рис. 7.1 ▾ Окно программы, демонстрирующей основные свойства компонента `TLabel`

```
begin
  Label1.ShowHint:=CheckBox1.Checked;
end;

// Установка/сброс прозрачного цвета.
procedure TForm1.CheckBox2Click(Sender: TObject);
begin
  Label1.Transparent:=CheckBox2.Checked;
end;
```

Компонент TEdit

Это тоже уже знакомый нам компонент. С его помощью мы вводили данные с клавиатуры. Ниже в табл. 7.3 приведены основные свойства этого компонента.

Таблица 7.3 ▼ Основные свойства компонента `TEdit`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Text`	Текст, отображаемый в поле компонента
`MaxLength`	Максимально допустимое число символов, которое можно ввести в поле компонента
`Font`	Шрифт, который используется для отображения текста
`Color`	Цвет текста, находящегося в поле компонента
`Left`	Расстояние от левой границы формы до левой границы компонента
`Top`	Расстояние от верхней границы формы до верхней границы компонента
`Width`	Ширина поля компонента
`Height`	Высота поля компонента
`BorderStyle`	Вид рамки компонента. Свойство может принимать значения `Single` (обычная рамка) или `None` (рамка отсутствует)
`ReadOnly`	Свойство, определяющее, будет ли использоваться компонент только для чтения. Если свойство установлено в `False`, то ввод данных разрешается, в противном случае ввод будет запрещен

Компоненты TButton и TBitBtn

Компонент `TButton` является последним из рассмотренных нами ранее компонентов. Он представляет собой командную кнопку. Свойства компонента приведены в табл. 7.4.

Таблица 7.4 ▾ Основные свойства компонента `TButton`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Caption`	Текст, отображаемый на кнопке
`Font`	Шрифт, который используется для отображения текста
`Left`	Расстояние от левой границы формы до левой границы компонента
`Top`	Расстояние от верхней границы формы до верхней границы компонента
`Width`	Ширина поля компонента
`Height`	Высота поля компонента
`Enabled`	Признак доступности кнопки. Кнопка доступна, если значение свойства равно `True`, если же оно равно `False` – кнопка недоступна
`Visible`	Признак видимости кнопки на поверхности формы. Если значение свойства равно `True` – кнопка отображается, в противном случае – кнопка невидима

Основным отличием компонента `TButton` от его аналога в .NET (компонента `Button`) является невозможность отображения картинки на кнопке.

Следующий компонент, который будет нами рассмотрен, – `TBitBtn` – также является командной кнопкой. Этот компонент более универсален, основное его отличие от предыдущего компонента заключается в том, что он может содержать картинку. Основные свойства компонента `TBitBtn` приведены в табл. 7.5.

Таблица 7.5 ▾ Основные свойства компонента `TBitBtn`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Caption`	Текст, отображаемый на кнопке
`Font`	Шрифт, используемый для отображения текста
`Left`	Расстояние от левой границы формы до левой границы компонента
`Top`	Расстояние от верхней границы формы до верхней границы компонента
`Width`	Ширина поля компонента
`Height`	Высота поля компонента
`Enabled`	Признак доступности кнопки. Кнопка доступна, если значение свойства равно `True`, если же оно равно `False` – кнопка недоступна
`Visible`	Признак видимости кнопки на поверхности формы. Если значение свойства равно `True`, кнопка отображается, в противном случае кнопка невидима
`Glyph`	Картинка, отображаемая на кнопке (файл изображения)
`NumGlyphs`	Количество картинок в файле изображения, указанного в свойстве `Glyph`
`Layout`	Свойство, определяющее взаимоположения картинки и текста на кнопке. Свойство может принимать следующие значения: `blGlyphLeft` – картинка располагается слева от надписи, `blGlyphRight` – справа, `blGlyphTop` – сверху, `blGlyphBottom` – снизу
`Spacing`	Расстояние от картинки до надписи, задаваемое в пикселях

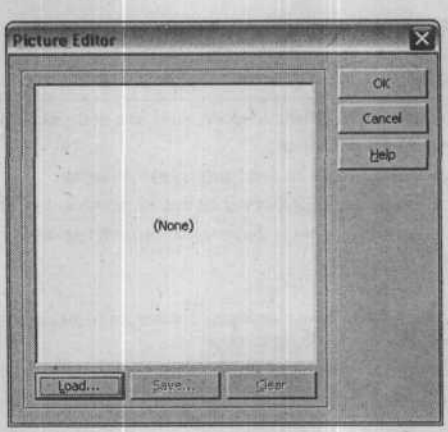

Рис. 7.2 ▼ Добавление картинки в кнопку `BitBtn`

Как видно из табл. 7.5, кнопка может содержать картинку. Добавить ее можно двумя способами – во время разработки формы и во время выполнения программы.

В первом случае необходимо в строке свойства `Glyph` окна **Object Inspector** нажать кнопку с тремя точками. После этого в появившемся окне **Picture Editor** (Редактор картинок) необходимо нажать на кнопку **Load** и указать файл формата `bmp` (рис. 7.2). Отметьте для себя, что в отличие от компонента `Button` для .NET никаких других форматов изображений не поддерживается.

Во втором случае надо воспользоваться методом `LoadFromFile`, указав в качестве параметра имя файла изображения:

```
BitBtn1.Glypth.LoadFromFile('example.bmp');
```

Компонент TImageList

Компонент `TImageList` представляет собой контейнер, содержащий набор картинок. Эти картинки могут быть использованы другими компонентами (например, `TToolBar`). Компонент не отображается в процессе выполнения программы, то есть является *невизуальным*. Если вы попробуете добавить компонент `TImageList` в проект, то он будет перенесен на форму, но в процессе работы программы вы его не увидите.

Основные свойства этого компонента приведены ниже в табл. 7.6.

Таблица 7.6 ▾ Основные свойства компонента `TImageList`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Width`	Ширина картинок коллекции
`Height`	Высота картинок коллекции

Набор картинок формируется во время разработки формы из заранее подготовленных картинок. Для добавления картинки в компонент необходимо выбрать его в окне **Design**, нажать правую кнопку мыши и в появившемся контекстном меню выбрать пункт **ImageList Editor** (Редактор списка картинок). Перед вами появится окно редактора свойств (рис. 7.3), в котором для добавления картинки в список следует нажать кнопку **Add**. В список можно добавлять картинки в формате bmp и ico. Все картинки должны быть *одного* размера и иметь *одинаковый* цвет фона.

При добавлении очередной картинки в список требуется также указать стиль ее отображения, который может быть трех типов:

- `Crop` – обычный стиль отображения;
- `Stretch` – растягивание (сжатие) картинки до размеров `Width` и `Height` компонента;
- `Center` – картинка центрируется без изменения размеров.

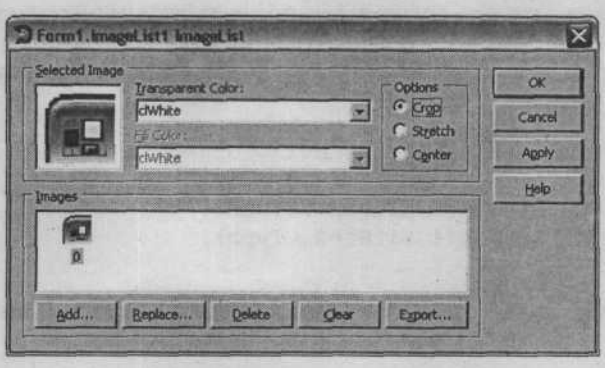

Рис. 7.3 ▾ Добавление картинок в коллекцию

Следует также обратить внимание на то, что список картинок объекта хранится в *файле ресурсов* проекта. Преобразование картинок осуществляется программой ImageList Editor, при этом исходные файлы картинок для работы программы не нужны.

Картинки в компонент TImageList можно добавить и программным путем. Для этого существует четыре метода, представленных в табл. 7.7.

Таблица 7.7 ▼ Добавление картинок в компонент TImageList во время выполнения программы

Метод	Описание
`function Add(Image: TBitmap, Mask: TBitmap): Integer;`	Добавляет в TImageList новую картинку, используя маску, указанную в параметре Mask. В качестве маски можно передавать Nil. Функция возвращает индекс добавленной картинки
`function AddMasked(Image: TBitmap, MaskColor: TColor): Integer;`	Добавляет в TImageList новую картинку, используя цвет, указанный в параметре MaskColor для создания маски. Функция возвращает индекс добавленной картинки
`function AddIcon(Image: TIcon): Integer;`	Добавляет в TImageList новую картинку, создавая ее из иконки. Так как иконка уже содержит маску, то ее указание не требуется. Функция возвращает индекс добавленной картинки
`procedure AddImages(Value: TCustomImageList);`	Копирует картинки из другого TImageList в текущий. В качестве источника картинок может быть передан указатель на любого наследника TCustomImageList, например TImageList

Кроме того, во время выполнения программы можно воспользоваться методом GetBitmap, который извлекает картинку в заранее созданный объект Tbitmap, и назначить его другому компоненту или отрисовать на форме или другом компоненте. Как уже говорилось, кнопка TBitBtn не поддерживает назначение картинки из TImageList, однако приведенный ниже пример позволяет это сделать:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ImageList1.GetBitmap(0,BitBtn1.Glyph);
  ImageList1.GetBitmap(1,BitBtn2.Glyph);
end;
```

Не забудьте, что индекс картинок в компоненте TImageList начинается с нуля.

Компонент TPanel

Компонент TPanel представляет собой контейнер для других компонентов и позволяет легко управлять компонентами, которые находятся на панели. Как

и в случае использования компонента Panel для .NET, компоненты, находящиеся на панели, наследуют свойства компонента TPanel. Например, чтобы сделать недоступными все компоненты на панели, достаточно присвоить значение False свойству Enabled панели. Свойства компонента TPanel приведены в табл. 7.8.

Таблица 7.8 ▾ Основные свойства компонента TPanel

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Caption	Текст в поле компонента
Color	Цвет поля компонента
Enabled	Свойство позволяет сделать доступными (Enabled=True) или недоступными (Enabled=False) все компоненты, которые размещены на панели
Visible	Свойство позволяет отображать (Visible=True) и скрывать (Visible=False) панель
Align	Определяет границу формы, к которой «прикреплена» панель. Панель может быть прикреплена к верхней (alTop), нижней (alBottom), левой (alLeft) или правой (alRight) границе формы, занимать всю форму (alClient) либо быть независимой (alCutsom)
Font	Задает шрифт панели. Все элементы, размещенные на панели, будут иметь указанный шрифт
Left	Расстояние от левой границы формы до левой границы компонента
Top	Расстояние от верхней границы формы до верхней границы компонента
Width	Ширина поля компонента
Height	Высота поля компонента

Из перечисленных в табл. 7.8 основных свойств более подробно следует остановиться на свойстве Align, позволяющем «прикрепить» панель к границе компонента, на котором расположена панель, например формы. В результате привязки панели к границе формы размер панели автоматически меняется при изменении размера компонента, на котором панель расположена.

Ниже на рис. 7.4 приведена форма окна программы, демонстрирующей некоторые особенности использования компонента TPanel.

На панель помещены различные компоненты. Все они наследуют свойства панели, влияющие на отрисовку, на которой располагаются. Программа позволяет изменить шрифт панели, а также сделать ее доступной или недоступной, видимой или невидимой. Соответственно, изменяются шрифт, доступность и видимость компонентов, расположенных на панели (листинг 7.2).

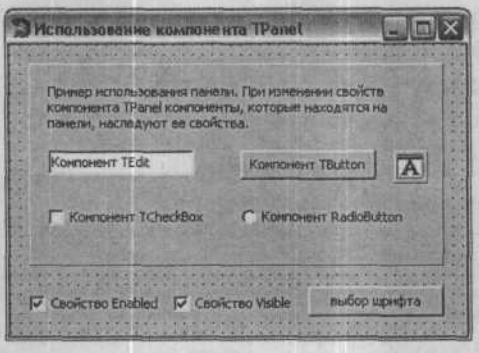

Рис. 7.4 ▼ Использование компонента `TPanel`

Листинг 7.2 ▼ Фрагмент программы, демонстрирующей особенности использования компонента `TPanel`

```
// Установка/сброс свойства Enabled панели.
procedure TForm1.CheckBox2Click(Sender: TObject);
begin
  Panel1.Enabled:=CheckBox2.Checked;
end;
// Установка/сброс свойства Visible панели.
procedure TForm1.CheckBox3Click(Sender: TObject);
begin
  Panel1.Visible:=CheckBox3.Checked;
end;

// Задание шрифта панели
// (используется компонент TFontDialog, расположенный
// на вкладке Dialogs).
procedure TForm1.Button2Click(Sender: TObject);
begin
  if FontDialog1.Execute then
    Panel1.Font:=FontDialog1.Font;
end;
```

Компонент TCheckBox

Компонент `TCheckBox` является переключателем (основные свойства приведены в табл. 7.9), который может находиться в одном из двух состояний: выбранном или невыбранном (иногда еще говорят – установленном или не установленном). Рядом с переключателем обычно находится поясняющий текст.

Таблица 7.9 ▾ Основные свойства компонента CheckBox

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Caption	Текст, располагающийся справа от кнопки
Checked	Свойство, определяющее, в каком состоянии находится переключатель. Если переключатель выбран, то значение свойства равно True, если не выбран – False
Enabled	Свойство, определяющее, доступен ли переключатель (Enabled=True) или нет (Enabled=False)
Visible	Свойство, позволяющее отображать (Visible=True) и скрывать (Visible=False) панель
Alignment	Взаимоположение кнопки и текста в поле компонента. Текст может располагаться справа (Aligment=taRightJustify) или слева (Aligment=taLeftJustify) от кнопки
AllowGrayed	Свойство определяет, будет ли переключатель иметь возможность находиться в трех состояниях. По умолчанию свойство равно False, что подразумевает два возможных состояния переключателя – установлен и не установлен

В предыдущих примерах мы достаточно много использовали подобный компонент – он служит для включения или выключения каких-либо элементов в окне программы.

Компоненты TRadioButton и TRadioGroup

Компоненты TRadioButton и TRadioGroup представляет собой группу кнопок (или переключателей) с поясняющим текстом, который обычно располагается справа. Можно сказать, что второй компонент является объединением компонентов TRadioButton и TGroupBox (описание см. ниже). Состояние кнопки зависит от состояния других кнопок (компонентов TRadioButton). В каждый момент времени в выбранном состоянии может находиться только одна из кнопок, находящихся на форме или в контейнере (например, кнопки, расположенные в разных компонентах TradioGroup, не зависят друг от друга). Возможна также ситуация, когда ни одна из кнопок не выбрана. В отличие от аналога компонента для .NET в поле компонента не может присутствовать картинка.

Несколько компонентов TRadioButton можно объединить в группу, разместив их в поле компонента TGroupBox. При этом состояние компонентов, принадлежащих одной группе, не зависит от состояния компонентов, принадлежащих другой группе.

В табл. 7.10 приведены основные свойства компонента TRadioButton.

Таблица 7.10 ▼ Основные свойства компонента TRadioButton

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Caption	Текст, располагающийся справа от кнопки
Checked	Свойство, определяющее, в каком состоянии находится переключатель. Если переключатель выбран, то значение свойства равно True, если не выбран – False
Enabled	Свойство, определяющее, доступен ли переключатель (Enabled=True) или нет (Enabled=False)
Visible	Свойство, позволяющее отображать (Visible=True) и скрывать (Visible=False) кнопку
Alignment	Взаимоположение кнопки и текста в поле компонента. Текст может располагаться справа (Aligment=taRightJustify) или слева (Aligment=taLeftJustify) от кнопки

Аналогичного результата можно добиться, если использовать компонент TRadioGroup. Для того чтобы добавить в компонент TRadioGroup кнопки, необходимо в окне **Object Inspector** щелкнуть в строке свойства Items по кнопке с тремя точками. После этого следует ввести несколько строк, содержание которых определит количество и названия кнопок TRadioButton в компоненте. Заполнить компонент кнопками можно и во время выполнения программы – для этого необходимо вызвать метод Add свойства Items:

```
RadioGroup1.Items.Add('новая кнопка');
```

Основные свойства компонента TRadioGroup приведены в табл. 7.11.

Таблица 7.11 ▼ Основные свойства компонента TRadioGroup

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Caption	Текст, располагающийся слева в верхней части панели
	Свойство, определяющее, в каком состоянии находится переключатель. Если переключатель выбран, то значение свойства равно True, если не выбран - False
Enabled	Свойство, определяющее, доступна ли панель: если Enabled=True – доступна, если Enabled=False – нет
Visible	Свойство, позволяющее отображать (Visible=True) и скрывать (Visible=False) панель
Items	Коллекция строк, определяющая количество и названия располагаемых в поле компонента кнопок TRadioButton
ItemIndex	Свойство, содержащее индекс нажатой в данный момент кнопки. Если ни одна кнопка не нажата, то значение равно -1

Если попробуете сделать две аналогичные панели (рис. 7.5), то заметите, что второй способ гораздо быстрее.

Попробуем написать простенькую программу, которая позволяет нам выбрать напиток и кондитерское изделие, – применим для сравнения оба варианта использования кнопок RadioButton. Форма окна программы приведена на рис. 7.5. В этой форме левая панель с кнопками реализована с

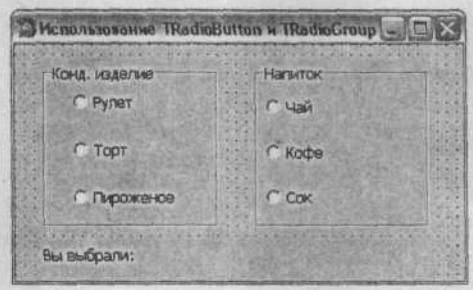

Рис. 7.5 ▾ Окно программы, демонстрирующей использование компонента `TRadioButton`

помощью компонентов `TGroupBox` и `TRadioButton`, правая – с помощью компонента `TRadioGroup`.

Текст программы приведен в листинге 7.3. Из особенностей использования компонента `TRadioGroup` можно отметить то, что событие `Click`, возникающее при нажатии на кнопку, является *общим* для всех кнопок панели. Для определения того, какая именно кнопка была нажата, необходимо использовать свойство `ItemIndex`.

Листинг 7.3 ▾ Использование компонентов `TRadioButton` и `TRadioGroup`

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    RadioGroup1: TRadioGroup;
    RadioButton1: TRadioButton;
    RadioButton2: TRadioButton;
    RadioButton3: TRadioButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure RadioGroup1Click(Sender: TObject);
    procedure RadioButton3Click(Sender: TObject);
    procedure RadioButton2Click(Sender: TObject);
    procedure RadioButton1Click(Sender: TObject);
  private
```

```
    { Private declarations }
  public
    { Public declarations }
  end;

const st = 'Вы выбрали: ';

var
  Form1: TForm1;
  ch1,ch2:string; // Результат выбора левой и правой части.

implementation

{$R *.dfm}

// Нажатие на кнопку 'Рулет'.
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
  ch1:='рулет';
  if ch2=''
    then Label1.Caption:=st+ch1
    else Label1.Caption:=st+ch1+' и '+ch2;
end;

// Нажатие на кнопку 'Торт'.
procedure TForm1.RadioButton2Click(Sender: TObject);
begin
  ch1:='торт';
  if ch2=''
    then Label1.Caption:=st+ch1
    else Label1.Caption:=st+ch1+' и '+ch2;
end;

// Нажатие на кнопку 'Пирожное'.
procedure TForm1.RadioButton3Click(Sender: TObject);
begin
  ch1:='пирожное';
  if ch2=''
    then Label1.Caption:=st+ch1
    else Label1.Caption:=st+ch1+' и '+ch2;
end;

// Для компонента TRadioGroup требуется определить,
// какая кнопка была нажата.
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0: ch2:='чай';  // Нажата кнопка 'Чай'.
    1: ch2:='кофе'; // Нажата кнопка 'Кофе'.
```

```
    2: ch2:='сок';  // Нажата кнопка 'Сок'.
  end;
  if ch1=''
    then Label1.Caption:=st+ch2
    else Label1.Caption:=st+ch1+' и '+ch2;
end;

// Начальные установки - при загрузке формы еще ничего не выбрано.
procedure TForm1.FormCreate(Sender: TObject);
begin
  ch1:='';
  ch2:='';
end;
end.
```

Компонент TGroupBox

Этот компонент мы уже упоминали ранее и даже использовали в предыдущих примерах. Как и свой аналог для .NET, компонент `TGroupBox` представляет собой контейнер для других компонентов. Обычно он используется для объединения компонентов в группы по функциональным признакам.

Основные свойства компонента приведены в табл. 7.12.

Таблица 7.12 ▾ Основные свойства компонента `TGroupBox`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Caption`	Текст, располагающийся в верхней части. Основное назначение свойства состоит в пояснении предназначения компонентов, находящихся в группе
`Font`	Шрифт контейнера и всех входящих в нее компонентов
`Enabled`	Свойство, определяющее, доступен ли компонент (`Enabled=True`) или нет (`Enabled=False`)
`Visible`	Свойство, позволяющее отображать (`Visible=True`) и скрывать (`Visible=False`) компонент
`Align`	Определяет границу формы, к которой «прикреплена» панель. Панель может быть прикреплена к верхней (`alTop`), нижней (`alBottom`), левой (`alLeft`) или правой (`alRight`) границе формы, занимать всю форму (`alClient`) либо быть независимой (`alCutsom`)

Компонент TComboBox

Компонент `TComboBox` представляет собой сочетание поля редактирования и списка, что позволяет вводить данные путем набора с клавиатуры или выбором из списка. Свойства компонента приведены в табл. 7.13.

Таблица 7.13 ▼ Основные свойства компонента `TComboBox`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Text`	Текст, располагающийся в поле для ввода (если компонент находится в режиме `csDropDown` или `csSimple`)
`Font`	Шрифт текста
`Enabled`	Свойство, определяющее, доступен ли компонент для ввода/выбора (`Enabled=True`) или нет (`Enabled=False`)
`Visible`	Свойство, позволяющее отображать (`Visible=True`) и скрывать (`Visible=False`) компонент
`Style`	Свойство, определяющее вид компонента: `csDropDown` – поле ввода и раскрывающийся список, `csSimple` – поле ввода со списком, `csDropDownList` – раскрывающийся список
`DropDownCount`	Максимальное количество показываемых элементов в выпадающем списке
`Items`	Элементы списка – коллекция строк
`ItemIndex`	Номер элемента, который в данный момент выбран в списке. Если выбранного элемента нет, то значение свойства равно -1
`Sorted`	*Признак необходимости сортировки элементов коллекции после добавления* очередного элемента

Список, отображаемый в поле компонента, можно формировать во время создания формы или во время работы программы.

Чтобы сформировать список во время работы программы, надо применить метод `Add` к свойству `Items`. Например, следующий фрагмент кода формирует упорядоченный по алфавиту список:

```
// Метод очистки элементов списка.
ComboBox1.Items.Clear
// Установка признака необходимости сортировки.
ComboBox1.Sorted:=True;
// Заполнение элементов списка.
ComboBox1.Text:='Петербург';
ComboBox1.Items.Add('Петербург');
ComboBox1.Items.Add('Москва');
ComboBox1.Items.Add('Новосибирск');
ComboBox1.Items.Add('Мурманск');
ComboBox1.Items.Add('Воронеж');
ComboBox1.Items.Add('Краснодар');
```

Аналогичного результата можно добиться, если использовать для ввода элементов списка окно редактора свойств **String List Editor**. С помощью этого окна список можно сформировать во время создания формы. Для этого необходимо щелкнуть на кнопке с тремя точками в строке свойства `Items` (Элементы) и в появившемся окне **String List Editor** (рис. 7.6) ввести элементы списка.

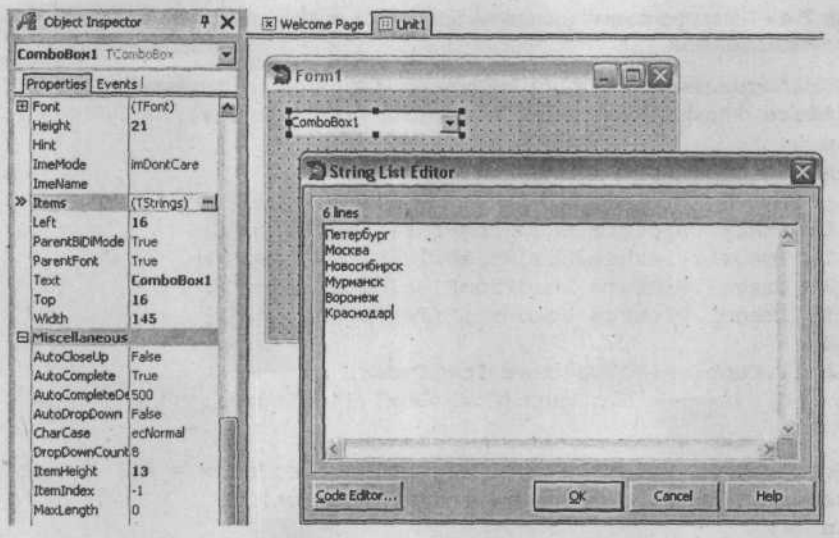

Рис. 7.6 ▾ Формирование списка компонента TComboBox во время создания формы

Рис. 7.7 ▾ Форма приложения, демонстрирующего вариант использования компонента TComboBox

Следующая программа демонстрирует использование компонента TComboBox для ввода данных. Форма приложения приведена на рис. 7.7.

Текст программы приведен в листинге 7.4.

Листинг 7.4 ▼ Текст программы (основных процедур), поясняющей использование компонента TComboBox

```
// Обрабатываем изменение текущего выбранного элемента.
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  case ComboBox1.ItemIndex of
    -1:;
    0: Image1.Picture.LoadFromFile('img01.jpg');
    1: Image1.Picture.LoadFromFile('img02.jpg');
    2: Image1.Picture.LoadFromFile('img03.jpg');
    3: Image1.Picture.LoadFromFile('img04.jpg');
  end;
  Label1.Caption:='Значение ItemIndex: '+
                  IntToStr(ComboBox1.ItemIndex);
end;

// Показываем, что изначально свойство ItemIndex = -1.
procedure TForm1.FormCreate(Sender: TObject);
begin
  Label1.Caption:='Значение ItemIndex: '+
                  IntToStr(ComboBox1.ItemIndex);
end;
```

Компонент TListBox

Данный компонент представляет собой список, в котором пользователь может выбирать нужный элемент. Основные свойства компонента приведены в табл. 7.14.

Таблица 7.14 ▼ Основные свойства компонента TListBox

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Items	Элементы списка – список строк
ItemIndex	Номер элемента, который в данный момент выбран в списке. Если выбранного элемента нет, то значение свойства равно –1
Items.Count	Общее количество элементов списка
Sorted	Признак необходимости (Sorted=True) сортировки элементов коллекции после добавления очередного элемента
MultiSelect	Свойство, определяющее режим выбора элементов списка. Может принимать следующие значения: True – в списке можно выбирать несколько элементов, False – только один элемент

Таблица 7.14 ▾ Основные свойства компонента `TListBox` (окончание)

Свойство	Комментарий
`ExtendedSelect`	Признак доступности расширенного выбора элементов списка в режиме `MultiSelect`. Если свойство установлено в `True`, то выбирать элементы списка можно с помощью клавиш **Shift** и **Ctrl**, в противном случае – только обычным способом
`Columns`	Задает количество колонок, в которые будет формироваться список, если элементы списка невозможно полностью отобразить в поле компонента
`Align`	Определяет границу формы, к которой «прикреплена» панель. Список может быть прикреплен к верхней (`alTop`), нижней (`alBottom`), левой (`alLeft`) или правой (`alRight`) границе формы, занимать всю форму (`alClient`) либо быть независимым (`alCutsom`)
`Font`	Шрифт, используемый для вывода элементов списка

Список, отображаемый в поле редактирования, можно сформировать как при создании формы, так и при выполнении программы. В первом случае необходимо щелкнуть по кнопке с тремя точками в поле свойства **Items** и в окне **String List Editor** (рис. 7.8) ввести элементы списка. Во втором случае необходимо применить уже знакомый нам метод `Add` свойства **Items**.

Далее приведена программа, которая демонстрирует пример использования этого компонента. Программа прослушивает несколько аудиофайлов в формате mp3. Для воспроизведения файлов используется компонент `TMediaPlayer` (находится на вкладке **System** окна **Tool Palette**), который в данной книге не рассматривается. Тем не менее в этом простом примере вы поймете основы его

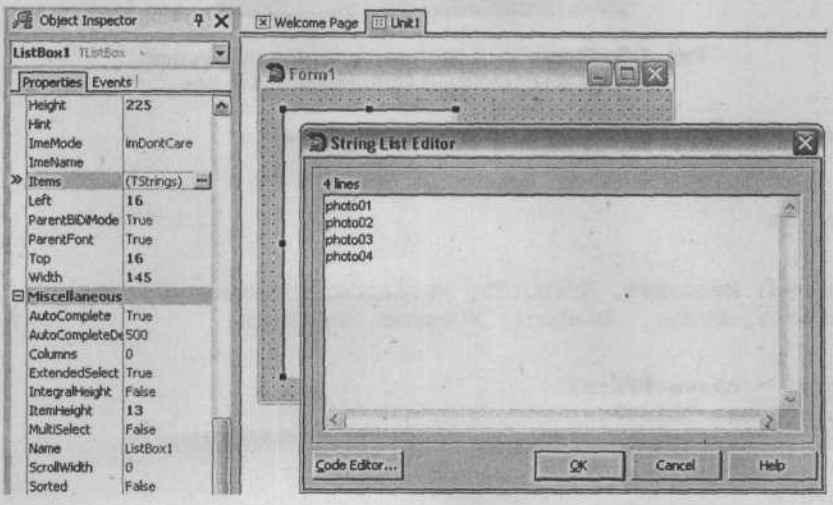

Рис. 7.8 ▾ Ввод элементов списка с помощью окна **String List Editor**

работы. Для того чтобы «проиграть» звуковой файл, необходимо определить имя файла, открыть его с помощью метода Open компонента TMediaPlayer и затем применить метод Play. Компонент TMediaPlayer в ходе выполнения программы не отображается (MediaPlayer1.Visible:=False). В данном примере вся последовательность действий воспроизведения звукового файла реализована как процедура, куда в качестве параметра передается имя файла, который необходимо воспроизвести. В свою очередь, имя файла получается с помощью компонента TListBox, в котором выбирается нужный элемент списка.

Форма окна программы приведена на рис. 7.9, а текст – в листинге 7.5.

Рис. 7.9 ▼ Форма окна программы, демонстрирующей вариант использования компонента TListBox

Листинг 7.5 ▼ Текст программы для просмотра изображений

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms,  Dialogs, MPlayer, StdCtrls;

type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    MediaPlayer1: TMediaPlayer;
    procedure ListBox1Click(Sender: TObject);
  private
    { Private declarations }
  public
```

```
    { Public declarations }
  end;

var
  Form1: TForm1;

// Перед использованием своей процедуры ее необходимо объявить.
procedure song_play(file_name:string);

implementation

{$R *.dfm}

// Процедура воспроизведения файла file_name.
procedure song_play(file_name:string);
begin
  Form1.MediaPlayer1.FileName:=file_name;
  Form1.MediaPlayer1.Open;
  Form1.MediaPlayer1.Play;
end;

// Процедура выбора элемента списка - определение
// имени воспроизводимого файла.
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  case ListBox1.ItemIndex of
    0: song_play('01.mp3');
    1: song_play('02.mp3');
    2: song_play('03.mp3');
    3: song_play('04.mp3');
  end;
end;

end.
```

Компонент TCheckListBox

Этот компонент также является списком, однако перед каждым его элементом находится переключатель TCheckBox. Свойства компонента приведены в табл. 7.15.

Таблица 7.15 ▾ Основные свойства компонента TCheckListBox

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Items	Элементы списка – коллекция строк
Items.Count	Общее количество элементов списка

Таблица 7.15 ▼ Основные свойства компонента TCheckListBox (окончание)

Свойство	Комментарий
Checked	Свойство, определяющее, выбран элемент списка или нет. Если список находится в режиме MultiSelect, то для проверки состояния элемента списка необходимо использовать его индекс (checked[index])
Sorted	Признак необходимости (Sorted=True) сортировки элементов коллекции после добавления очередного элемента
Columns	Задает количество колонок, в которые будет формироваться список, если элементы списка невозможно полностью отобразить в поле компонента
Align	Определяет границу формы, к которой «прикреплена» панель. Панель может быть прикреплена к верхней (alTop), нижней (alBottom), левой (alLeft) или правой (alRight) границе формы, занимать всю форму (alClient) либо быть независимой (alCustom)
MultiSelect	Свойство, определяющее режим выбора элементов списка. Может принимать следующие значения: True – в списке можно выбирать несколько элементов, False – только один элемент
Font	Шрифт, используемый для отображения содержимого поля редактирования и элементов списка

Формируется список TCheckedListBox также двумя способами – либо с использованием окна **String List Editor**, либо с использованием метода Add к свойству Items. Доступ к переключателям во время выполнения программы можно получить через свойство Checked:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i : integer;
  total : integer;
begin
  total := 0;
  for i := 0 to CheckListBox1.Items.Count-1 do
    if CheckListBox1.Checked[i] then inc(total);
  ShowMessage('Всего включено '+IntToStr(total)+' флажков');
end;
```

Компонент TImage

Компонент TImage является аналогом компонента PictureBox (.NET) и обеспечивает отображение иллюстрации (файла рисунка). Основные свойства компонента приведены в табл. 7.16.

Таблица 7.16 ▼ Основные свойства компонента TImage

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам

Таблица 7.16 ▾ Основные свойства компонента `TImage` (окончание)

Свойство	Комментарий
`Picture`	Иллюстрация, отображаемая в поле компонента
`Center`	Признак необходимости размещения картинки независимо от ее размера по центру поля компонента. Если свойство установлено в `True`, то картинка центрируется
`Stretch`	Признак необходимости изменения размеров изображения под размер поля компонента. Если свойство установлено в `True`, то картинка «вписывается» в поле компонента, ее ширина и высота становятся равными значениям свойств `Width` и `Height` поля компонента соответственно. Также при этом осуществляется масштабирование
`Proportional`	Признак необходимости изменения размеров изображения под размер поля компонента. В отличие от свойства `Stretch` производится пропорциональное масштабирование по одной из осей (в зависимости от размера картинки и размера компонента)
`Transparent`	Признак необходимости задания прозрачности компонента. Если свойство установлено в `True`, то фон картинки (прозрачный цвет) не отображается
`AutoSize`	Признак необходимости автоматического изменения размеров компонента под размер загружаемой картинки. Если свойство установлено в `True`, то при загрузке изображения в компонент значения его свойств `Width` и `Height` автоматически изменятся
`Left`	Расстояние от левой границы формы до левой границы компонента
`Top`	Расстояние от верхней границы формы до верхней границы компонента
`Width`	Ширина поля компонента (области отображения)
`Height`	Высота поля компонента (области отображения)
`Align`	Определяет границу формы, к которой «прикреплена» панель. Панель может быть прикреплена к верхней (`alTop`), нижней (`alBottom`), левой (`alLeft`) или правой (`alRight`) границе формы, занимать всю форму (`alClient`) либо быть независимой (`alCustom`)
`Visible`	Признак необходимости отображения компонента (и, соответственно, изображения)

Чтобы задать иллюстрацию во время обработки формы, необходимо в строке свойства `Picture` щелкнуть по кнопке с тремя точками и в появившемся окне редактора свойств **Picture Editor** выбрать файл изображения. Добавленное таким образом в форму изображение будет обработано Delphi и помещено в файл ресурсов проекта. Соответственно, сам файл изображения нам больше не понадобится. Если же потребуется загружать картинку во время выполнения программы, то следует воспользоваться методом `LoadFromFile` (дословно – «загрузить из файла»). В качестве параметра нужно указать полное имя загружаемого в `TImage` файла. Например, инструкция

```
Image1.Picture.LoadFromFile('d:\example.bmp');
```

загрузит и отобразит в компоненте `Image1` файл d:\example.bmp. Компонент может отображать не только файлы формата bmp. Данным методом можно также загружать файлы jpg, wmf, emf. Однако следует учесть один момент. Если вы захотите, например, воспользоваться методом `LoadFromFile` для загрузки jpg-файла, то будет выдана ошибка. Все дело в том, что для загрузки и

отображения таких файлов (jpg) необходимо в секции uses «прописать» модуль jpeg. А если вы воспользовались окном **Picture Editor** для загрузки jpg-файла, то этого делать не требуется – ссылка на модуль jpeg в секцию uses будет помещена автоматически.

Помните, что при загрузке изображения компонент TImage не обеспечивает масштабирования без искажения, если размер компонента (области отображения рисунка) непропорционален размеру рисунка.

На рис. 7.10 приведена форма окна программы, поясняющей возможные режимы отображения картинок компонентом TImage.

Рис. 7.10 ▼ Форма окна программы, поясняющей использование компонента TImage

Текст основных процедур программы приведен в листинге 7.6.

Листинг 7.6 ▼ Использование различных режимов отображения компонента Timage

```
// Сброс/установка свойства Stretch.
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  Image1.Stretch:=CheckBox1.Checked;
end;
```

```
// Сброс/установка свойства Center.
procedure TForm1.CheckBox2Click(Sender: TObject);
begin
  Image1.Center:=CheckBox2.Checked;
end;
// Сброс/установка свойства Proportional.
procedure TForm1.CheckBox3Click(Sender: TObject);
begin
  Image1.Proportional:=CheckBox3.Checked;
end;
```

Компонент TUpDown

Этот компонент предназначен для ввода числовых данных. Компонент находится на вкладке **Win32** окна **Tool Palette** и предназначен для ввода *только* целых числовых данных. Вводить данные можно как с клавиатуры, так и с использованием стрелок **Увеличить/Уменьшить**.

Основные свойства компонента приведены в табл. 7.17.

Таблица 7.17 ▾ Основные свойства компонента `TUpDown`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Position`	Текущее числовое значение компонента
`Min`	Минимальное значение, которое можно ввести в компонент
`Max`	Максимальное значение, которое можно ввести в компонент
`Increment`	Величина, на которую увеличивается (или уменьшается) значение свойства `Position` при нажатии на соответствующую стрелку
`Wrap`	Свойство определяет, будет ли сбрасываться значение `Position` при достижении границы диапазона (`Min` или `Max`). Если, например, свойство установлено в `False`, то по достижении минимума или максимума при дальнейшем нажатии соответствующих стрелок изменение значения свойства `Position` происходить не будет
`Orientation`	Свойство задает тип ориентации кнопок-стрелок **Увеличить/Уменьшить**. Может принимать следующие значения: `udVertical` – вертикальная ориентация, `udHorizontal` – горизонтальная

Ниже приведен пример использования компонента `TUpDown` для ввода числовых данных. На рис. 7.11 изображена форма окна программы, которая позволяет с помощью данного компонента изменять свойства компонента `TMemo`.

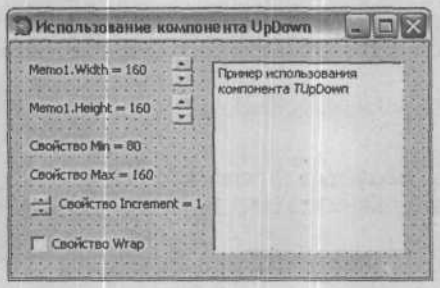

Рис. 7.11 ▼ Форма окна программы, демонстрирующей вариант использования компонентов `TUpDown`

Компонент `TMemo` очень похож на компонент `TEdit`, но является более универсальным, так как позволяет хранить множество строк текста, определяемое свойством `Lines`. Фактически компонент `TextBox` (.NET) является совокупностью `TMemo` и `TEdit` по функциональным возможностям. С помощью компонентов `UpDown1` и `UpDown2` (листинг 7.7) изменяются размеры компонента `Memo1`, а с помощью компонента `UpDown3` – свойство `Increment` компонентов `UpDown1` и `UpDown2`.

Листинг 7.7 ▼ Пример использования компонента `TUpDown`

```
// Нажатие на компонент UpDown3.
// Изменение приращения значения (свойство Increment).
procedure TForm1.UpDown3Click(Sender: TObject; Button: TUDBtnType);
begin
  UpDown1.Increment:=UpDown3.Position;
  UpDown2.Increment:=UpDown3.Position;
  Label3.Caption:='Свойство Increment = '+
                  IntToStr(UpDown3.Position);
end;

// Нажатие на компонент UpDown2.
procedure TForm1.UpDown2Click(Sender: TObject; Button: TUDBtnType);
begin
  Memo1.Width:=UpDown2.Position;
  Label4.Caption:='Memo1.Width = '+IntToStr(UpDown2.Position);
end;

// Нажатие на компонент UpDown1.
procedure TForm1.UpDown1Click(Sender: TObject; Button: TUDBtnType);
begin
  Memo1.Height:=UpDown1.Position;
  Label5.Caption:='Memo1.Height = '+IntToStr(UpDown1.Position);
end;
```

```
// Сброс/установка свойства Wrap для UpDown1 и UpDown2.
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  UpDown1.Wrap:=CheckBox1.Checked;
  UpDown2.Wrap:=CheckBox1.Checked;
end;
```

Компонент TStatusBar

Компонент TStatusBar представляет собой область (панель) для вывода служебной информации. Обычно такая панель располагается в нижней части программы и может разбиваться на несколько частей. Данный компонент можно найти на вкладке **Win32** окна **Tool Palette**.

Свойства компонента TStatusBar приведены в табл. 7.18.

Таблица 7.18 ▾ Основные свойства компонента TStatusBar

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Panels	Коллекция панелей – объектов типа StatusBarPanel
Align	Свойство, определяющее местоположение панели. Панель может располагаться внизу окна (alBottom), вверху окна (alTop), быть прижатой к левому (alLeft) или правому (alRight) краю, заполнять все пространство окна (alClient) или же размещаться по желанию программиста в любом месте окна (alNone). По умолчанию панель всегда прижата к нижней части окна (alBottom)
SizeGrip	Признак необходимости отображать кнопки изменения размера панели (три наклонные черты) в правом нижнем углу панели
SimplePanel	Свойство определяет режим, в котором отображается компонент. Если свойство установлено в True, то отображается «простая» панель с текстом, указанным в свойстве SimpleText. Если свойство установлено в False, то отображается коллекция панелей (объекты типа StatusBarPanel, определенные в свойстве Panels)
SimpleText	Текст, который отображается в панели, если она не разделена на области (объекты типа StatusBarPanel)
BorderWidth	Толщина обрамления (границы) панели. Свойство указывается в пикселях
Font	Шрифт, которым выводится текстовая информация панелей (в любом режиме отображения)

Для того чтобы разделить панель на несколько составных частей, необходимо в строке свойства Panels (Панели) нажать кнопку с тремя точками и в появившемся окне **Editing StatusBar1.Panel** щелкнуть на кнопке **Add** столько раз, сколько панелей планируется использовать (рис. 7.12). В этом же окне можно выполнить настройку каждой из панелей. Свойства выбранной (редактируемой в данный момент) панели отображены в правой части окна **Editing StatusBar1.Panel**.

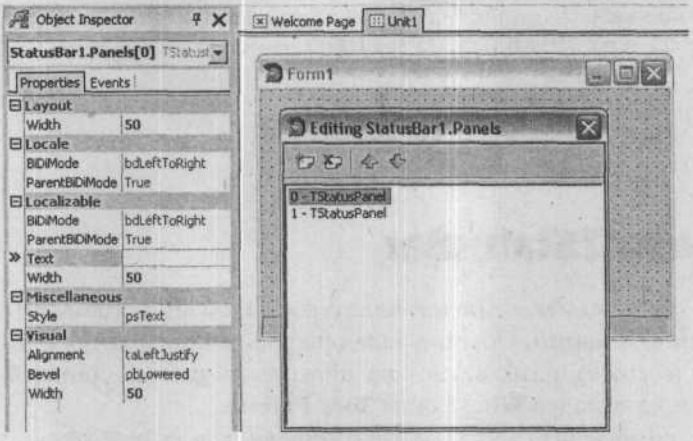

Рис. 7.12 ▼ Использование окна **Editing StatusBar1.Panel**

Основные свойства панелей объекта `TStatusBarPanel` приведены в табл. 7.19.

Таблица 7.19 ▼ Основные свойства панелей объекта `TStatusBarPanel`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Text`	Текст, отображаемый в панели
`Width`	Максимальная ширина панели. Свойство указывается в пикселях
`Alignment`	Свойство, задающее выравнивание текста в панели. Может принимать значение `taLeftJustify` (по левому краю), `taRightJustify` (по правому краю) или `taCenter` (по центру)
`Bevel`	Стиль обрамления панели. Панель может быть «утоплена» (`pbLowered`), не иметь стиля (`pbNone`) или быть выпуклой (`pbRaised`)

Следующая программа, окно которой приведено на рис. 7.13, демонстрирует вариант применения этого компонента.

Текст программы (основных процедур) приведен ниже в листинге 7.8.

Листинг 7.8 ▼ Использование компонента `TStatusBar` (основные процедуры)

```
// Процедура обработки события OnTimer.
// Событие возникает через каждые 100 мс.
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  // Обновляем содержимое панелей.
  // Вывод даты.
  // Вывод времени.
```

Рис. 7.13 ▾ Форма окна программы, поясняющей вариант использования компонента `TStatusBar`

```
  StatusBar1.Panels[0].Text:='Текущая дата: '+DateToStr(Date);
  StatusBar1.Panels[1].Text:='Текущее время: '+TimeToStr(Time);
end;

// Запуск таймера.
procedure TForm1.FormCreate(Sender: TObject);
begin
  Timer1.Enabled:=True;
end;
```

Компонент TTimer

Компонент `TTimer` представляет собой компонент, генерирующий последовательность событий `OnTimer`. Данный компонент невидим во время выполнения программы, то есть является *невизуальным*. Свойства компонента `TTimer` приведены в табл. 7.20.

Таблица 7.20 ▾ Основные свойства компонента `TTimer`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Enabled`	Свойство, определяющее, включен таймер или нет
`Interval`	Свойство, определяющее промежуток времени, через который происходит генерация события `Tick`. Задается в миллисекундах

Следующая программа (рис. 7.14), которая является простейшим секундомером, поясняет использование компонента `TTimer`.

Сначала свойство `Enabled` компонента `TTimer` установлено в `False`, поэтому таймер не генерирует никаких сообщений. Процедура обработки события `Click` на кнопке `Button1` присваивает свойству `Enabled` значение `True`, тем самым запуская таймер. Процедура обработки события `Tick` отсчитывает интервалы с момента нажатия на кнопку **Пуск**. Нажатие на кнопку **Стоп** останавливает секундомер –

Рис. 7.14 ▼ Форма программы и использованием компонента `TTimer`

опять же изменением свойства `Enabled` таймера `Timer1`. Текст программы приведен в листинге 7.9.

Листинг 7.9 ▼ Текст программы, поясняющей использование компонента `TTtimer`

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Button1: TButton;
    Button2: TButton;
    Timer1: TTimer;
    procedure Button2Click(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  min,sec:integer; // Минуты/секунды.

implementation

{$R *.dfm}

// Процедура обработки события Click для левой кнопки ('Запуск/
// Стоп').
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Timer1.Enabled
    then
      // Если секундомер работает,
      begin
        Timer1.Enabled:=False;   // то останавливаем таймер.
        Button1.Caption:='Пуск'; // Меняем название кнопки
                                 // со 'Стоп' на 'Пуск'.
        Button2.Enabled:=True;   // Кнопка 'Сброс' теперь
                                 // доступна.
      end
    else
      // Если секундомер не работает,
      begin
        Timer1.Enabled:=True;    // то запускаем таймер.
        Button1.Caption:='Стоп'; // Меняем название кнопки
                                 // с 'Пуск' на 'Стоп'.
        Button2.Enabled:=False;  // Кнопка 'Сброс' теперь
                                 // недоступна.
      end;
end;

// Процедура обработки события таймера.
procedure TForm1.Timer1Timer(Sender: TObject);
var str:string;
begin
  if sec = 59 then       // Если количество секунд равно 59,
    begin                // то увеличиваем минуты на 1,
      inc(min);          // а секунды обнуляем.
      sec := 0;
    end                 // В противном случае просто
    else                // наращиваем секунды.
      inc(sec);
    // Формируем строку в формате m:ss.
    str := str + IntToStr(sec);
    if Length(str) = 1 then str := '0' + str;
    // Создаем эффект 'мигания' двоеточия
    // (отображаем двоеточие только на четных секундах).
    if sec mod 2 = 0
      then str := ':' + str
      else str := ' ' + str;
    str := IntToStr(min) + str;
    Panel1.Caption := str;
end;

// Процедура обработки события Click для правой кнопки ('Сброс').
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  sec := 0;
  min := 0;
  Panel1.Caption := '0:00';
end;
end.
```

Компонент TToolBar

Этот компонент представляет собой панель инструментов, на которой можно размещать *командные кнопки*. Свойства компонента TToolBar приведены в табл. 7.21.

Таблица 7.21 ▼ Основные свойства компонента TToolBar

Свойство	Комментарий
Name	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
Color	Цвет поля компонента
Transparent	Прозрачность компонента. Если свойство равно True, то панель прозрачна, иначе – нет
Visible	Свойство, позволяющее отображать (Visible=True) и скрывать (Visible=False) панель
Align	Определяет границу формы, к которой «прикреплена» панель инструментов. Панель инструментов может быть прикреплена к верхней (alTop), нижней (alBottom), левой (Left) или правой (alRight) границе формы, занимать всю форму (alClient) либо быть независимой (alCutsom)
Font	Задает шрифт панели. Все элементы, размещенные на панели, будут иметь указанный шрифт
Left	Расстояние от левой границы формы до левой границы компонента
Top	Расстояние от верхней границы формы до верхней границы компонента
Width	Ширина поля компонента
Height	Высота поля компонента
Enabled	Свойство, позволяющее сделать доступными (Enabled=True) или недоступными (Enabled=False) все компоненты, размещенные на панели
ButtonHeight	Высота кнопок на панели инструментов
ButtonWidth	Ширина кнопок на панели инструментов
Images	Ссылка на компонент TImageList, содержащий картинки для кнопок
Flat	Определяет,является ли панель «плоской». Если свойство установлено в True, то кнопки не выделяются на панели, пока на них не наведена мышь, в противном случае граница кнопок всегда видна
Wrapable	Определяет, может ли (Wrapable=True) панель инструментов переносить часть кнопок на следующую строку, если на первой строке не хватает места для размещения всех элементов, или нет (Wrapable=False)

В качестве рекомендаций по использованию этого компонента можно отметить следующие. Перед использованием компонента лучше сначала добавить к форме компонент `TImageList` (см. выше) и произвести его настройку. После того как картинки для кнопок будут определены, можно настраивать и саму панель с кнопками. Для этого нужно выделить компонент `TToolBar`, нажать правую кнопку и в появившемся контекстном меню (рис. 7.15) выбрать пункт **New Button** (Новая кнопка). После того как нужное количество кнопок будет добавлено к панели, можно производить настройки для каждой кнопки. Для этого необходимо выделить нужную кнопку, и в окне Object Inspector отобразятся ее свойства.

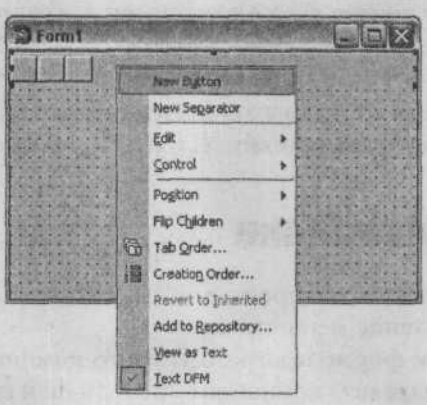

Рис. 7.15 ▾ Добавление кнопок к панели `TToolBar`

Необходимо отметить, что в отличие от компонента `ToolBar` для .NET, в котором при нажатии на панель возникает событие, общее для всех кнопок, в компоненте `TToolBar` для Win32 такое событие возникает для *каждой* кнопки. Соответственно, для каждой кнопки придется отдельно описывать процедуру обработки события `ToolButtonClick`.

Кроме того, кроме кнопок на панели инструментов `TToolBar` можно разместить любые другие компоненты, например `TComboBox`.

Компонент TProgressBar

Этот компонент полностью идентичен индикатору `ProgressBar`, рассмотренному нами в главе 5, и обычно используется для отображения протекания какого-либо процесса. Такими процессами, например, могут быть копирование файлов, загрузка данных и т.п. Чтобы не раздражать людей кажущимся бездействием программы (которая на самом деле выполняет какие-либо

трудоемкие операции), обычно используют именно этот компонент. Свойства компонента представлены в табл. 7.22.

Таблица 7.22 ▼ Основные свойства компонента `TProgressBar`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Position`	Значение, отображаемое в поле компонента в виде прямоугольников, количество которых пропорционально значению свойства `Position`
`Min`	Минимальное значение, которое может принимать свойство `Position`
`Max`	Максимальное значение, которое может принимать свойство `Position`
`Step`	Свойство, определяющее приращение свойства `Position`. Используется для изменения свойства `Position` методом `StepIt`

Обратите внимание на то, что выход значения свойства `Position` за границы диапазона, определяемого значениями `Min` и `Max`, вызовет ошибку.

Компонент TMainMenu

Этот компонент, аналогом которого в .NET служит компонент `MainMenu`, представляет собой главное меню программы.

После добавления к форме программы этого компонента необходимо его настроить. Для этого выделите компонент `TMainMenu` и нажмите правую кнопку мыши. В появившемся контекстном меню выберите пункт **Menu Designer** (Конструктор меню), и перед вами появится окно (рис. 7.16), которое предстоит заполнить элементами меню.

Для создания элемента меню необходимо выполнить щелчок в области ввода текста и ввести название пункта меню. Как только вы что-нибудь наберете, справа и внизу появятся области для ввода следующих элементов меню. Заполняя эти области, вы постепенно создаете структуру вашего меню (рис. 7.17).

Пункты меню можно отделять друг от друга с помощью разделителей. Для этого необходимо навести курсор в то место, куда вы хотите поместить разделитель, и вместо названия пункта меню указать символ вычитания (-). После того как вы нажмете клавишу **Enter**, в указанное вами место будет вставлен разделитель (горизонтальная черта).

Сформировав структуру, можно приступать к настройке главного меню. Для этого нам потребуется ознакомиться с основными свойствами компонента `TMainMenu`, точнее говоря – со свойствами пунктов нашего меню (`TMenuItem`).

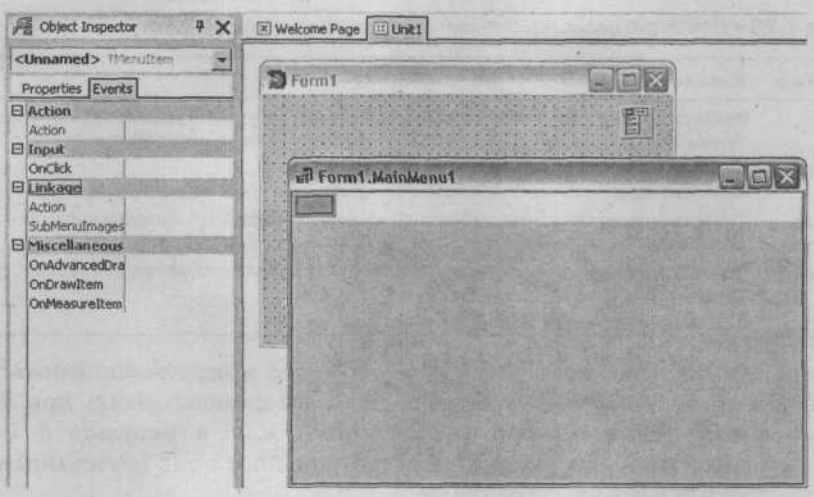

Рис. 7.16 ▾ Окно редактора меню **Menu Designer**

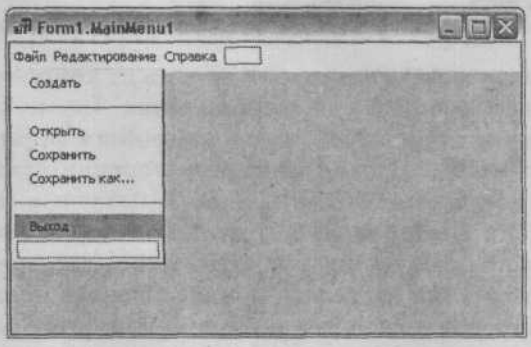

Рис. 7.17 ▾ Заполнение элементами меню компонента `TMainMenu`

Доступ к свойствам элемента меню `TMenuItem` можно получить из окна **Object Inspector**, нажав на один из пунктов меню, который собираемся настроить. Итак, основные свойства, которые мы будем настраивать, приведены в табл. 7.23.

Таблица 7.23 ▾ Основные свойства элементов меню `TMainMenu`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Caption`	Название элемента меню

Таблица 7.23 ▼ Основные свойства элементов меню `TMainMenu` (окончание)

Свойство	Комментарий
`Enabled`	Признак доступности элемента меню. Если значение свойства равно `False`, то название пункта изображается инверсным цветом и при нажатии на него событие `Click` не происходит
`Bitmap`	Свойство задает картинку формата bmp, отображаемую слева от пункта меню
`ShortCut`	Свойство, определяющее комбинацию клавиш (или клавишу), нажатие которой расценивается как выбор соответствующего пункта меню
`Visible`	Признак необходимости скрывать (`Visible=False`) или отображать (`Visible=True`) пункт меню
`Images`	Ссылка на список картинок для элементов меню

Теперь поговорим о событии `Click`, которое может воспринимать наше меню. Событие возникает в результате щелчка по элементу меню, при нажатии клавиши **Enter** (если выбран пункт меню) или в результате нажатия функциональной клавиши, указанной в свойстве `Shortcut` (сочетание клавиш быстрого доступа к этому элементу меню).

Компонент TPopupMenu

Этот компонент, аналогом которого в .NET служит компонент `ContextMenu`, также представляет собой контекстное меню. После добавления этого компонента на форму приложения в строке свойств формы появится новое свойство – `TPopupMenu`. Для определения перечня пунктов меню нужно выделить компонент, нажать правую кнопку мыши и в контекстном меню выбрать пункт **Menu Designer** (Конструктор меню). После этого появится аналогичное рассмотренному на рис. 7.16 окно, где требуется заполнить элементы меню так же, как мы делали это для главного меню.

После того как контекстное меню будет создано, следует выполнить его окончательную настройку – задать значения свойств пунктов меню `TMenuItem`, а также определить процедуры обработки событий. В отличие от компонента `TMainMenu` для `TPopupMenu` необходимо *дополнительно* определить компонент, для которого это меню создано. Для этого в свойство `PopupMenu` компонента необходимо поместить ссылку на контекстное меню. Свойства объекта `TPopupMenu` приведены в табл. 7.24.

Таблица 7.24 ▼ Основные свойства элемента меню `TPopupMenu`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`Caption`	Название элемента меню

Таблица 7.24 ▾ Основные свойства элемента меню TPopupMenu (окончание)

Свойство	Комментарий
Enabled	Признак доступности элемента меню. Если значение свойства равно False, то название пункта изображается инверсным цветом и при нажатии на него событие Click не происходит
Bitmap	Свойство задает картинку формата bmp, отображаемую слева от пункта меню
ShortCut	Свойство, определяющее комбинацию клавиш (или клавишу), нажатие которых расценивается как выбор соответствующего пункта меню
Visible	Признак необходимости скрывать (Visible=False) или отображать (Visible=True) пункт меню
Images	Ссылка на список картинок для элементов меню

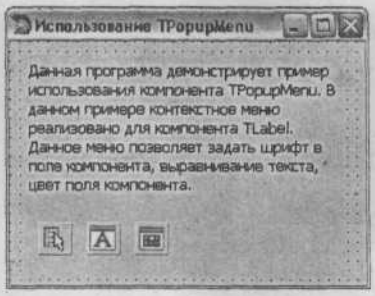

Рис. 7.18 ▾ Форма программы, демонстрирующей вариант использования компонента TPopupMenu

Текст программы приведен в листинге 7.10.

Листинг 7.10 ▾ Текст программы (основные процедуры), демонстрирующей вариант использования компонента TPopupMenu

```
// Выбор пункта меню 'Выбор шрифта'.
procedure TForm1.N1Click(Sender: TObject);
begin
  FontDialog1.Execute;
  Label1.Font:=FontDialog1.Font;
end;

// Выбор пункта меню 'Выравнивание/по левому краю'.
procedure TForm1.N3Click(Sender: TObject);
begin
  Label1.Alignment:=taLeftJustify;
end;

// Выбор пункта меню 'Выравнивание/по правому краю'.
```

```
procedure TForm1.N4Click(Sender: TObject);
begin
  Label1.Alignment:=taRightJustify;
end;

// Выбор пункта меню 'Выравнивание/по центру'.
procedure TForm1.N5Click(Sender: TObject);
begin
  Label1.Alignment:=taCenter;
end;

// Выбор пункта меню 'Выбор цвета'.
procedure TForm1.N6Click(Sender: TObject);
begin
  if ColorDialog1.Execute then
    Label1.Color:=ColorDialog1.Color;
end;

// Выбор пункта меню 'Исходные установки'.
procedure TForm1.N8Click(Sender: TObject);
begin
  // Восстанавливаем шрифт.
  Label1.Font.Color:=clWindowText;
  Label1.Font.Name:='Tahoma';
  Label1.Font.Size:=10;
  Label1.Font.Style:=[];
  // Восстанавливаем цвет.
  Label1.Color:=clBtnFace;
  // Восстанавливаем выравнивание текста.
  Label1.Alignment:=taLeftJustify;
end;
```

Компонент TOpenDialog

Данный компонент представляет собой стандартное диалоговое окно, позволяющее выбирать (открывать) файлы. Хотя некоторые свойства этого компонента нам уже знакомы, тем не менее приведу основные из них (табл. 7.25).

Таблица 7.25 ▼ Основные свойства компонента `TOpenDialog`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам

Таблица 7.25 ▾ Основные свойства компонента `TOpenDialog` (окончание)

Свойство	Комментарий
`FileName`	Имя выбранного пользователем файла. Помимо собственно имени файла содержит также и путь к каталогу, в котором находится файл
`Filter`	Свойство определяет описание и фильтр, по которому будут отображаться файлы. В списке файлов отображаются только те файлы, описание которых соответствует заданной маске. Например, значение свойства `Текстовые файлы | *.txt` будет отображать только текстовые файлы с соответствующим расширением
`FilterIndex`	Если фильтр имеет несколько элементов (например, `Текстовые файлы | *.txt | Файлы изображений |*.jpg`), то значение свойства определяет тот фильтр, который используется в момент появления окна на экране
`InitialDir`	Имя каталога, содержимое которого будет отображаться при появлении окна
`DefaultExt`	Расширение файла, указываемое по умолчанию
`Title`	Заголовок диалогового окна

Отображение самого диалогового окна обеспечивает метод `Execute`. Результатом завершения будет код клавиши, которая была нажата пользователем (**OK** или **Cancel**). Помните, что данный компонент не выполняет непосредственно открытия файла, – его назначение состоит в том, чтобы получить имя файла, над которым будут производиться соответствующие действия (например, чтение файла).

Из приведенных в табл. 7.25 свойств можно дополнительно отметить свойство `Filter`. Для более удобного задания фильтра можно воспользоваться специальным редактором фильтров. Для этого в поле свойства `Filter` необходимо нажать на кнопку с тремя точками, после чего на экране появится окно редактора свойств **Filter Editor** (рис. 7.19).

В левой части окна вводится название фильтра, в правой – сам фильтр, по которому будет производиться отображение необходимых файлов.

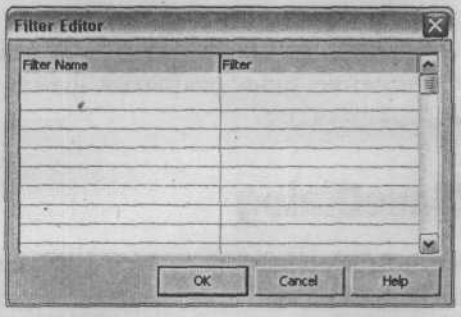

Рис. 7.19 ▾ Окно редактора фильтра **Filter Editor**

Следующая программа использует компонент TOpenDialog для открытия текстового файла, который затем отображается с помощью компонента TMemo. Форма окна программы приведена на рис. 7.20, текст программы – в листинге 7.11.

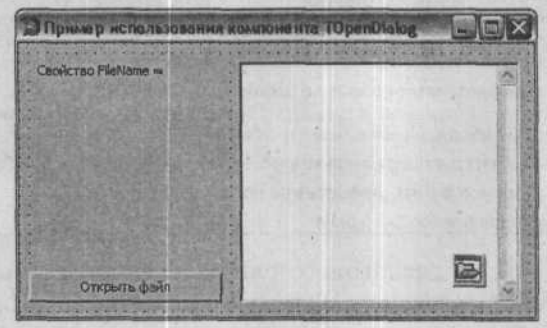

Рис. 7.20 ▼ Форма программы, использующей компонент TOpenDialog

Листинг 7.11 ▼ Текст программы (процедуры обработки события Click), демонстрирующей использование компонента TOpenDialog

```
// Нажатие на кнопку 'Открыть файл'.
procedure TForm1.Button1Click(Sender: TObject);
begin
  // Открываем файл и заполняем компонент Memo текстом.
  if OpenDialog1.Execute then
    Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
  // Выводим имя открытого файла.
  Label2.Caption:='Свойство FileName = '+OpenDialog1.Filename;
end;
```

Диалоговое окно открытия файла появляется в результате нажатия кнопки **Открыть файл**. Отображение окна открытия файла осуществляет метод Execute компонента TOpenDialog.

Компонент TSaveDialog

Этот компонент также предназначен для работы с файлами (служит для сохранения файлов). Основные свойства компонента TSaveDialog приведены в табл. 7.26.

Таблица 7.26 ▾ Основные свойства компонента `TSaveDialog`

Свойство	Комментарий
`Name`	Имя компонента, используемое в программе для доступа к компоненту и его свойствам
`FileName`	Имя выбранного пользователем файла. Помимо собственно имени файла содержит также и путь к каталогу, в котором находится файл
`Filter`	Свойство, определяющее описание и фильтр, по которому будут отображаться файлы. В списке файлов отображаются только те файлы, описание которых соответствует заданной маске. Например, значение свойства `Текстовые файлы | *.txt` будет отображать только текстовые файлы с соответствующим расширением
`FilterIndex`	Если фильтр имеет несколько элементов (например, `Текстовые файлы | *.txt | Файлы изображений |*.jpg`), то значение свойства определяет тот фильтр, который используется в момент появления окна на экране
`InitialDir`	Имя каталога, содержимое которого будет отображаться при появлении окна
`DefaultExt`	Расширение файла, указываемое по умолчанию
`Title`	Заголовок диалогового окна

Отображение диалогового окна сохранения файла обеспечивает уже знакомый нам метод `Execute`, значением которого является `True`, если пользователь нажал кнопку **OK**. Приведенная ниже программа демонстрирует основные возможности этого компонента.

При успешном завершении диалога (нажатии кнопки **OK**) формируется текстовый файл, в который записывается содержимое компонента `TMemo` (рис. 7.21).

Текст программы (процедура обработки события `Click` на кнопке **Сохранить текст в файл**) приведен в листинге 7.12.

Рис. 7.21 ▾ Форма окна программы, демонстрирующей использование `TSaveDialog`

Листинг 7.12 ▼ Текст программы, демонстрирующей использование компонента TSaveDialog

```
// Нажатие на кнопку 'Сохранить текст в файл'.
procedure TForm1.Button1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
    Memo1.Lines.SaveToFile(SaveDialog1.FileName);
end;
```

Изучаем основы работы с графикой в Win32

Эта глава посвящена описанию работы с графикой при создании программ для Win32. Изложение материла в главе построено по той же схеме, что и описание работы с графикой для .NET, поэтому при изложении я буду иногда ссылаться на вопросы, рассмотренные в главе 6.

Итак, переходим к изучению основных вопросов, касающихся работы с графикой в Win32.

Создавая программу, вы можете использовать специальный компонент, отображающий картинки, – `TImage`. Кроме этого, можно рисовать непосредственно на поверхности формы, а также любого компонента, имеющего соответствующую поверхность для рисования. Поверхности формы (и любого объекта, на котором можно рисовать) соответствует объект `Canvas`, методы которого и обеспечивают вывод графических примитивов (точек, линий, эллипсов, многоугольников и т.д.). В любом случае, отображаете ли вы графический файл или рисуете самостоятельно, вам будет необходимо использовать объект `Canvas` (то есть применить соответствующие методы). Рисовать в Win32 очень просто – достаточно написать процедуру обработки события `FormPaint` (Рисование).

В качестве примера приведу текст программы (листинг 8.1), которая обрабатывает это событие и выводит на поверхность формы картинку, а также некоторые простейшие графические примитивы.

Листинг 8.1 ▾ Пример обработки события FormPaint

```
procedure TForm1.FormPaint(Sender: TObject);
var
```

```
  img:TBitmap;
begin
  // Отображение картинки.
  img:=TBitmap.Create;
  img.LoadFromFile('win32.bmp');
  Canvas.Draw(10,10,img);
  // Отображение надписи.
  Font.Name:='Arial';
  Font.Size:=14;
  Canvas.Brush.Color:=Form1.Color;
  Canvas.TextOut(200,10,'Графические возможности');
  Canvas.TextOut(200,40,'Delphi for Win32');
  // Рисуем прямоугольные области.
  Canvas.Brush.Style:=bsClear;
  Canvas.Rectangle(20,100,100,200);
  Canvas.Rectangle(30,110,110,210);
  Canvas.Rectangle(40,120,120,220);
  // Рисуем закрашенные прямоугольные области.
  Canvas.Brush.Style:=bsSolid;
  Canvas.Brush.Color:=clRed;
  Canvas.Rectangle(200,100,300,200);
  Canvas.Brush.Color:=clGreen;
  Canvas.Rectangle(240,110,340,210);
  Canvas.Brush.Color:=clBlue;
  Canvas.Rectangle(280,120,380,220);
end;
```

Результат работы программы приведен на рис. 8.1.

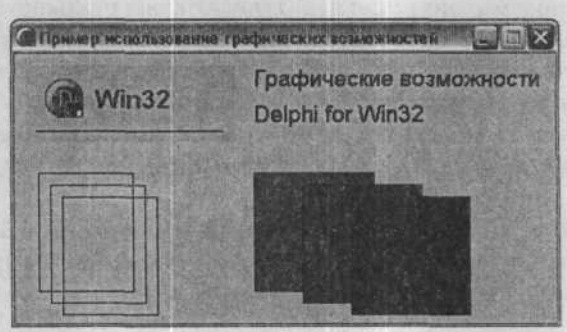

Рис. 8.1 ▼ Использование основных графических возможностей при обработке события `FormPaint`

Холст – графическая поверхность для рисования

Методы рисования графических примитивов рассматривают свойство `Canvas` как некоторую поверхность (аналог объекта `Graphics` в .NET), *холст*, на котором они могут рисовать.

Итак, эти методы используют так называемую *графическую* поверхность (холст), которая состоит из точек (пикселей). Положение каждой точки характеризуется двумя координатами: горизонтальной (X) и вертикальной (Y).

Координаты отсчитываются от левого верхнего угла (рис. 8.2). Максимальный размер, который может быть использован для рисования на окне программы, хранят свойства `ClientsWidth` и `ClientHeight` формы.

Далее мы познакомимся с основными инструментами, позволяющими рисовать на поверхности формы.

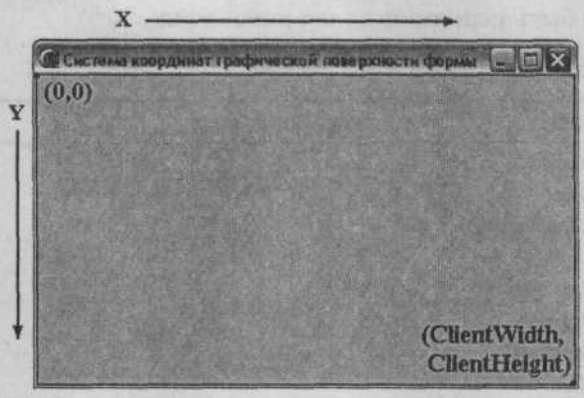

Рис. 8.2 ▾ Система координат точек графической поверхности формы

Карандаш и кисть – основные инструменты для рисования

Рисование осуществляется уже известными нам специальными инструментами – карандашами или кистями. Карандаш позволяет задать вид линии (толщина, цвет, стиль, режим отображения), а кисть – способ закраски определенной области (цвет и стиль закраски).

Карандаш

Объект «карандаш» (`Canvas.Pen`) определяет вид линии – толщину, цвет, стиль и режим отображения. Обычно карандаш используется для вычерчивания точек, линий и контуров. Ниже в табл. 8.1 приведены свойства объекта `TPen`.

Таблица 8.1 ▼ Основные свойства объекта `TPen`

Свойство	Комментарий
`Color`	Задает цвет линии (контура) – константа типа `TColor`
`Mode`	Определяет режим отображения (задается специальными константами)
`Style`	Вид линии (задается специальными константами)
`Width`	Свойство определяет толщину линии в пикселях

Как видно из табл. 8.1, значение свойства `Color` задает цвет вычерчиваемого объекта. Далее в табл. 8.2 приведены основные варианты цвета, который может быть использован для рисования.

Таблица 8.2 ▼ Некоторые варианты значения свойства `Pen.Color`

Константа	Цвет
`clBlack`	Черный
`clWhite`	Белый
`clSilver`	Серебристый
`clGray`	Серый
`clMaroon`	Коричневый
`clRed`	Красный
`clGreen`	Зеленый
`clBlue`	Синий
`clYellow`	Желтый
`clLime`	Салатный
`clAqua`	Бирюзовый

Кроме того, вы можете использовать так называемые системные цвета – цвета графических элементов операционной системы. В этом случае цвет вычерчиваемых объектов будет жестко привязан к текущей цветовой схеме операционной системы.

Значение свойства `Width` задает толщину выводимой линии в пикселях. Например, инструкция

```
Canvas.Pen.Width:=4;
```

задает линию толщиной в 4 пикселя.

Свойство Style определяет вид линии. Линия может быть непрерывной или прерывистой, состоящей из штрихов различной длины. В табл. 8.3 приведены примеры констант (значений свойства Style), задающих тот или иной вид линии.

Таблица 8.3 ▾ Значения констант, определяющих вид линии

Константа	Вид вычерчиваемой линии
psSolid	Сплошная линия
psDot	Пунктирная линия с короткими штрихами
psDash	Пунктирная линия с длинными штрихами
psDashDot	Пунктирная линия с чередованием длинных и коротких штрихов
psDashDotDot	Пунктирная линия с чередованием длинного и двух коротких штрихов
psClear	Неотображаемая линия (используется, например, в случаях, когда не надо отображать границу области)

При задании констант, определяющих вид линий, помните, что если свойство Width больше единицы, то пунктирные линии выводятся как сплошные.

Свойство Mode задает так называемый режим отображения. Фактически оно определяет, как будет формироваться цвет точек линии в зависимости от цвета точек холста, через которые эта линия проходит. По умолчанию цвет линии определяется значением свойства Color, однако программист может задать несколько режимов отображения, которые приведены в табл. 8.4.

Таблица 8.4 ▾ Значения констант, задающих различные режимы отображения линии

Константа	Цвет линии
pmBlack	Черный цвет линии, не зависит от цвета, определяемого свойством Color
pmWhite	Белый цвет линии, не зависит от цвета, определяемого свойством Color
pmNot	Цвет точки является инверсным по отношению к цвету холста, в который выводится точка линии
pmCopy	Цвет линии определяется значением свойства Color
pmNotCopy	Цвет линии определяется как инверсный по отношению значению свойства Color

Ниже приведен пример использования объекта TPen. В этом примере приведена процедура обработки события FormPaint, которая рисует несколько фигур с использованием объекта TPen.

```
// Использование установок карандаша.
procedure TForm1.FormPaint(Sender: TObject);
```

```
begin
  Canvas.Pen.Width:=2;
  Canvas.Pen.Color:=clRed;
  Canvas.Ellipse(100,150,10,20);
  Canvas.Pen.Width:=3;
  Canvas.Pen.Color:=clBlue;
  Canvas.Rectangle(200,120,150,20);
  Canvas.Pen.Color:=clGreen;
  Canvas.Ellipse(120,60,100,10);
  Canvas.Pen.Width:=1;
  Canvas.Pen.Color:=clBlack;
  Canvas.Pen.Style:=psDot;
  Canvas.Rectangle(90,160,120,180);
end;
```

Кисть

Инструмент «кисть» (`Canvas.Brush`) используется для закраски внутренних областей геометрических фигур. Например, фрагмент кода следующего вида

```
Canvas.Brush.Style:=bsSolid;
Canvas.Brush.Color:=clRed;
Canvas.Rectangle(10,10,200,150);
```

рисует закрашенный прямоугольник красного цвета.

Основные свойства кисти показаны в табл. 8.5.

Таблица 8.5 ▼ Основные свойства объекта `TBrush`

Константа	Цвет линии
`Color`	Цвет закраски области
`Style`	Стиль заполнения области

Свойство `Style` задает стиль заполнения области. Стиль области определяет, каким образом будет заштрихована (закрашена) область. Значения констант, задающих стиль заполнения области, приведены в табл. 8.6.

Таблица 8.6 ▼ Константы, задающие стиль заполнения (закраски области)

Константа	Цвет линии
`bsSolid`	Сплошная заливка области
`bsClear`	Область не закрашивается
`bsHorizontal`	Область заполняется горизонтальной штриховкой
`bsVertical`	Область заполняется вертикальной штриховкой

Таблица 8.6 ▾ Константы, задающие стиль заполнения (закраски области) (окончание)

Константа	Цвет линии
bsCross	Область заполняется штриховкой в клетку
bsFDiagonal	Область заполняется диагональной штриховкой с наклоном линий вперед
bsBDiagonal	Область заполняется диагональной штриховкой с наклоном линий назад
bsDiagCross	Область заполняется диагональной штриховкой в клетку

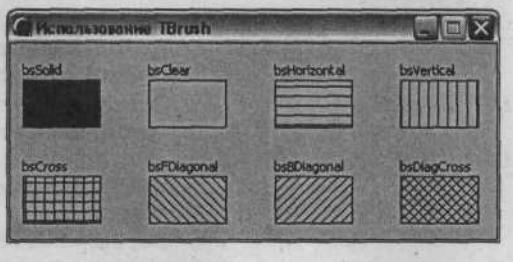

Рис. 8.3 ▾ Пример использования кисти

Далее приведен пример программы, демонстрирующий использование кисти. Программа, окно которой приведено на рис. 8.3, выводит несколько прямоугольных областей, заполненных различными вариантами штриховки.

Текст программы приведен в листинге 8.2.

Листинг 8.2 ▾ Текст программы, демонстрирующей вариант использования кисти

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls;

type
  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
```

```
  Form1: TForm1;
implementation
{$R *.dfm}
// Использование установок кисти.
procedure TForm1.FormPaint(Sender: TObject);
const
  st_name:array [1..8] of string =
    ('bsSolid','bsClear','bsHorizontal',
     'bsVertical','bsCross','bsFDiagonal',
     'bsBDiagonal','bsDiagCross');  // Имена стилей отображения.
var
  x,y:integer;                      // Координаты вывода
                                    // прямоугольника.
  _width,_height:integer;           // Длина и высота
                                    // прямоугольника.
  br_style:TBrushStyle;             // Тип используемой для
                                    // закраски кисти.
  k:integer;                        // Номер стиля кисти.
  i,j:integer;                      // Переменные-счетчики циклов
                                    // для отображения
                                    // восьми прямоугольников.
begin
  y:=30;
  _width:=65;
  _height:=40;
  for i:=1 to 2 do
    begin
      x:=10;
      for j:=1 to 4 do
        begin
          // k задает номер стиля.
          k:=j+(i-1)*4;
          case k of
            1:br_style:=bsSolid;
            2:br_style:=bsClear;
            3:br_style:=bsHorizontal;
            4:br_style:=bsVertical;
            5:br_style:=bsCross;
            6:br_style:=bsFDiagonal;
            7:br_style:=bsBDiagonal;
            8:br_style:=bsDiagCross;
          end;
          // Рисуем закрашенный прямоугольник.
          Canvas.Brush.Color:=clBlack;
```

```
            Canvas.Brush.Style:=br_style;
            Canvas.Rectangle(x,y,x+_width,y+_height);
            // Подписываем название стиля.
            Canvas.Brush.Style:=bsClear;
            Canvas.TextOut(x,y-15,st_name[k]);
            x:=x+_width+40;
          end;
        y:=y+_height+40;
      end;
end;

end.
```

Изучаем основные графические примитивы

Любой создаваемый нами рисунок представляет собой совокупность графических *примитивов* – точек, линий, окружностей, прямоугольников и т.п. Для изображения подобных примитивов используются соответствующие методы, некоторые из которых приведены в табл. 8.7.

Таблица 8.7 ▾ Методы вычерчивания основных графических примитивов

Метод	Действие
`MoveTo(x,y)`	Перемещает в указанную позицию (`x`, `y`) карандаш. Не вычерчивает никаких графических примитивов
`LineTo(x,y)`	Рисует линию. Рисование осуществляется из текущей позиции карандаша в позицию (`x`, `y`)
`PolyLine(P_Array)`	Рисует ломаную линию. Метод работает аналогично, в качестве параметров в данном случае передается массив точек (`x`, `y`) – узловых точек ломаной линии
`Rectangle(x1,y1,x2,y2)`	Рисует прямоугольник. Параметры `x1`, `y1`, `x2`, `y2` – координаты находящихся на одной диагонали углов прямоугольника
`RoundRect(x1,y1,x2,y2,x3,y3)`	Рисует скругленный прямоугольник. Первые четыре параметра аналогичны, пара (`x3`, `y3`) задает эллипс, четверть которого используется для скругления
`FillRect(P_Rect)`	Рисует закрашенный прямоугольник. В качестве параметра передается структура типа `TRect` – координаты точек, ограничивающих прямоугольную область
`FrameRect(P_Rect)`	Рисует контур прямоугольника. В качестве параметра передается структура типа `TRect` – координаты точек, ограничивающих прямоугольную область
`Polygon(P_Array)`	Рисует многоугольник. В качестве параметров в данном случае передается массив точек (`x`, `y`) – узловых точек ломаной линии
`Ellipse(x1,y1,x2,y2)`	Рисует эллипс. Параметры `x1`, `y1`, `x2`, `y2` – координаты диагональных углов области, внутри которой вычерчивается эллипс

Таблица 8.7 ▼ Методы вычерчивания основных графических примитивов (окончание)

Метод	Действие
`Arc(x1,y1,x2,y2,x3,y3,x4,y4)`	Рисует дугу. Параметры `x1, y1, x2, y2` – задают область, в которую вписан эллипс, `(x3, y3)` – начальную точку дуги, `(x4, y4)` – конечную точку дуги
`Pie(x1,y1,x2,y2,x3,y3,x4,y4)`	Рисует сектор. Параметры аналогичны параметрам метода `Arc`
`Pixels[x,y]`	Свойство для доступа к точкам (пикселям). Меняя цвет точек, можно «рисовать» на поверхности

Далее подробно разберем методы вычерчивания графических примитивов.

Линия

Для рисования прямой линии используется метод `LineTo`. В качестве параметров метода используются координаты точки, в которую из текущего местоположения карандаша должна быть проведена линия:

```
<Компонент>.Canvas.LineTo(x,y)
```

Например, мы хотим нарисовать линию из точки `(0,0)` в точку `(100,100)`. Тогда нарисовать нашу линию можно инструкцией:

```
Form1.Canvas.MoveTo(0,0);
Form1.Canvas.LineTo(100,100);
```

Ломаную линию можно нарисовать, используя метод `PolyLine`. В общем виде инструкция рисования ломаной линии выглядит следующим образом:

```
<Компонент>.Canvas.PolyLine(P_array);
```

где `P_array` – массив узловых точек типа `TPoint`. Каждый элемент этого массива представляет собой пару координат узловой точки `(x, y)`.

В качестве примера использования данных методов приведу программу, отображающую динамику (график) изменения стоимости ценных бумаг с течением времени. Предполагается, что массив `price` содержит информацию о стоимости в определенные моменты времени.

Текст программы приведен в листинге 8.3.

Листинг 8.3 ▼ Текст программы, поясняющей использование методов `LineTo` и `PolyLine`

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs;
```

```
type
  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormPaint(Sender: TObject);
const
  // Массив значений подписей к оси X.
  lbl1:array [1..7] of string = ('понедельник',
                                 'вторник',
                                 'среда',
                                 'четверг',
                                 'пятница',
                                 'суббота',
                                 'воскресенье');
var
  x0,y0:integer;     // Центр координат.
  dx,dy:integer;     // Шаг координатной сетки.
  x,y:integer;       // Текущие значения координат.
  _height:integer;   // Высота области отображения.
  _width:integer;    // Ширина области отображения.
  lbl_Y:real;        // Оцифровка осей координат (ось Y).
  dlbl_Y:integer;    // Шаг оцифровки (ось Y).
  i:integer;
  price:array[1..7] of TPoint; // Массив значений стоимости ценных бумаг.
begin
  // Определение начальных значений переменных.
  x0:=40; y0:=320; // Положение центра координат в (40,320).
  dx:=80; dy:=40;  // Шаг по оси X = 80, по оси Y = 40.
  lbl_Y:=3200;     // Минимальное значение стоимости.
  dlbl_Y:=100;     // Шаг приращения значения стоимости.
  _height:=300;    // Высота координатной сетки (ось Y).
  _width:=600;     // Длина координатной сетки  (ось X).
  // Заполняем массив значений стоимости ценных бумаг (точек TPoint).
  for i:=1 to 7 do price[i].x:=40+80*i; // Координата x.
  // Это значения функции - стоимость, координата y.
```

```
price[1].y:=140;
price[2].y:=180;
price[3].y:=160;
price[4].y:=260;
price[5].y:=120;
price[6].y:=180;
price[7].y:=200;
// ----------
// Выполняем рисование.
with Form1.Canvas do
  begin
    Pen.Style:=psSolid;
    Pen.Width:=1;
    Pen.Color:=clBlack;
    // Рисуем оси координат.
    Moveto(x0,y0);
    LineTo(x0,y0-_height);
    Moveto(x0,y0);
    LineTo(x0+_width,y0);
    // Рисуем подписи к оси X и вспомогательную сетку.
    x:=x0+dx;
    for i:=1 to 7 do
      begin
        MoveTo(x,y0-3);LineTo(x,y0+3);
        TextOut(x-15,y0+10,lbl1[i]);
        Pen.Style:=psDot;
        MoveTo(x,y0-3);LineTo(x,y0-_height);
        Pen.Style:=psSolid;
        x:=x+dx;
      end;
    // Рисуем подписи к оси Y и вспомогательную сетку.
    y:=y0-dy;
    for i:=1 to 7 do
      begin
        MoveTo(x0-3,y);LineTo(x0+3,y);
        TextOut(x0-25,y,FloatToStr(lbl_Y));
        Pen.Style:=psDot;
        MoveTo(x0+3,y);LineTo(x0+_width,y);
        Pen.Style:=psSolid;
        y:=y-dy;
        lbl_Y:=lbl_Y+dlbl_Y;
      end;
    // Рисуем график функции.
    Pen.Width:=3;
    Pen.Color:=clRed;
```

```
      MoveTo(x0+dx,y0-dy);
      for i:=1 to 7 do PolyLine(price);
    end;
end;

end.
```

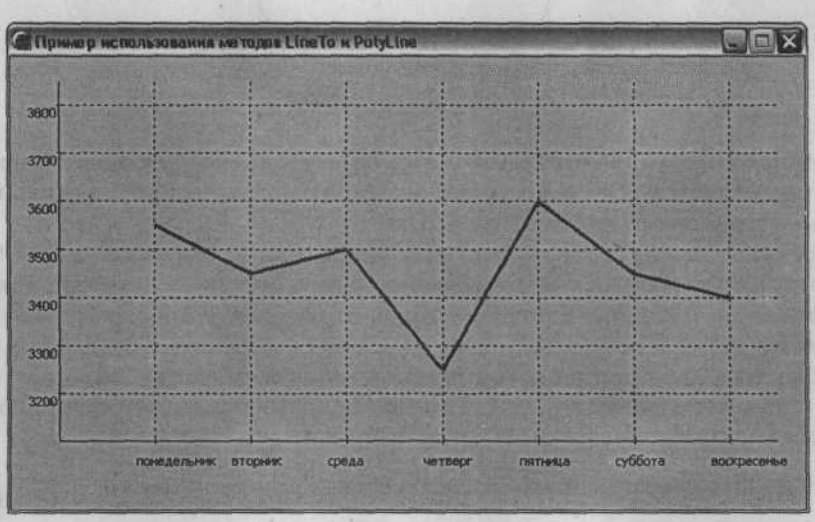

Рис. 8.4 ▾ Результат работы программы с использованием методов `LineTo` и `PolyLine`

Результат работы программы показан на рис. 8.4.

Прямоугольник

Прямоугольник можно начертить методом `Rectangle`. В качестве параметров этого метода указываются координаты двух углов, расположенных на одной диагонали:

```
<Компонент>.Canvas.Rectangle(x1,y1,x2,y2);
```

Методом `RoundRect` также можно начертить прямоугольник. Отличие данного метода от предыдущего состоит в том, что прямоугольник рисуется со скругленными углами. Инструкция вызова метода `RoundRec` выглядит следующим образом:

```
<Компонент>.Canvas.RoundRect(x1,y1,x2,y2,x3,y3);
```

где `x3` и `y3` – параметры, которые определяют «скругленность» углов прямоугольника. На рис. 8.5 можно увидеть, что параметры `x3` и `y3` задают размер эллипса, четверть которого используется для рисования скругленного угла.

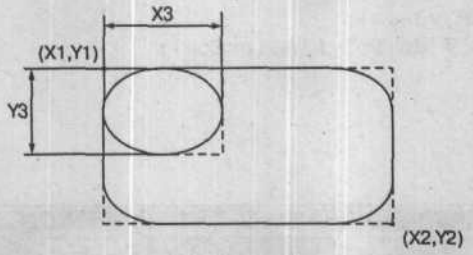

Рис. 8.5 ▼ Значение параметров при вызове метода `RoundRect`

При рисовании прямоугольника вид линии контура (ее цвет, ширина и стиль) определяется текущими установками карандаша (`TPen`), а стиль и цвет заливки – установками кисти (`TBrush`).

Существуют также и другие методы рисования прямоугольников. Метод `FillRect` рисует закрашенный прямоугольник, метод `FrameRect` – контур прямоугольника. Отличие данных методов от рассмотренных ранее состоит в том, что им передается только один параметр – структура типа `Rect`. Поля структуры содержат координаты прямоугольной области, которые заполняются с помощью функции `Rect`. Пример использования методов рисования прямоугольников приведен ниже в листинге 8.4.

Листинг 8.4 ▼ Пример использования методов рисования прямоугольников

```
procedure TForm1.FormPaint(Sender: TObject);
var
  rec1,rec2:TRect;  // Структуры для рисования методами FillRect и
                    // FrameRect.
begin
  // Рисуем прямоугольники с помощью методов Rectangle и RoundRect.
  with Form1.Canvas do
    begin
      Brush.Color:=clMaroon;
      Rectangle(20,20,100,80);
      Brush.Color:=clBlue;
      RoundRect(120,20,200,80,20,10);
    end;
  // Заполняем поля структур Rect.
  rec1:=Rect(20,120,100,180);
  rec2:=Rect(120,120,200,180);
  // Рисуем прямоугольники с помощью методов FillRect и FrameRect.
  with Form1.Canvas do
    begin
```

```
      Brush.Color:=clRed;
      FillRect(rec1);
      Brush.Color:=clBlue;
      FrameRect(rec2);
    end;
end;
```

Результат работы программы можно увидеть на рис. 8.6.

Рис. 8.6 ▾ Результат применения различных методов рисования прямоугольников

Многоугольник

Многоугольник можно нарисовать методом `Polygon`. При этом вычерчивается контур многоугольника. Инструкция рисования многоугольника выглядит следующим образом:

```
<Компонент>.Canvas.Polygon(M_points);
```

Параметр `M_points` представляет собой массив из элементов типа `TPoint`. Эти элементы являются узловыми вершинами (так же, как и в случае рисования ломаной линии). Вершина, которая является последней в массиве, соединяется с первой вершиной.

Цвет и стиль границы многоугольника определяются значениями свойств карандаша (`TPen`), а цвет и стиль заливки – свойствами кисти (`TBrush`). Например, следующий фрагмент кода (листинг 8.5) вычерчивает правильный восьмиугольник с использованием метода `Polygon`.

Листинг 8.5 ▾ Использование метода `Polygon`

```
procedure TForm1.FormPaint(Sender: TObject);
var x0,y0:integer;    // Координаты геометрического центра фигуры.
    angle:integer;
```

```
    dangle:integer;  // Угол, образуемый между соседними вершинами.
    radius:integer;  // Радиус окружности, в которую вписана фигура.
    n:integer;       // Число вершин фигуры.
    dx,dy:integer;
    P_Array:array[1..8] of TPoint; // Массив с координатами 8
                                   // вершин.
    i:integer;       // Счетчик цикла.
begin
  // Выполняем начальные установки.
  x0:=135;y0:=100;
  n:=8;
  radius:=50;
  dangle:=round(360/n);  // Определяем угол между двумя соседними
                         // вершинами.
  angle:= 90;            // Начальный угол = 90.
  // Определяем в цикле координаты (x,y) вершин многоугольника.
  for i:=1 to n do
    begin
      dx:=round(radius*cos(PI*angle/180));
      dy:=round(radius*sin(PI*angle/180));
      angle:=angle+dangle;
      P_Array[i].x:=x0+dx;
      P_array[i].y:=y0-dy;
    end;
  // Рисуем правильный восьмиугольник.
  Form1.Canvas.Brush.Color:=clRed;
  Form1.Canvas.Polygon(P_Array);
end;
```

А на экране результат выполнения данного кода будет выглядеть так, как показано на рис. 8.7.

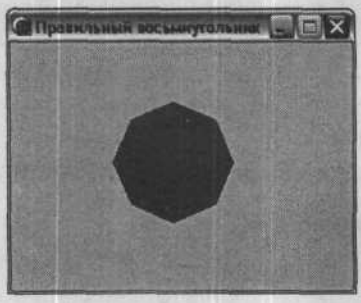

Рис. 8.7 ▼ Применение метода `Polygon`

Окружность и эллипс

Эллипс можно нарисовать методом `Ellipse`. Этот же метод рисует и окружность, которая является частным случаем эллипса. В качестве параметров метода указывается прямоугольная область, ограничивающая фигуру (рис. 8.8):

```
<Компонент>.Canvas.Ellipse(x1,y1,x2,y2);
```

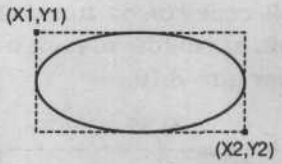

Рис. 8.8 ▾ Способ задания эллипса

Цвет и стиль границы эллипса определяются значениями свойств карандаша (`TPen`), а цвет и стиль заливки – свойствами кисти (`TBrush`).

Дуга

Дуга представляет собой часть эллипса. Поэтому в методе `Arc`, который позволяет нарисовать дугу, в качестве параметров необходимо дополнительно указать две точки, определяющие начало и конец дуги. Инструкция вызова метода `Arc` выглядит следующим образом:

```
<Компонент>.Canvas.Arc(x1,y1,x2,y2,x3,y3,x4,y4);
```

Параметры `x1`, `y1`, `x2`, `y2` определяют эллипс, частью которого является дуга. Параметры (`x3`, `y3`) задают начальную точку дуги, (`x4`, `y4`) – конечную точку дуги (рис. 8.9).

Необходимо заметить, что дуга вычерчивается *против* часовой стрелки, от начальной точки (`x3`, `y3`) к конечной (`x4`, `y4`).

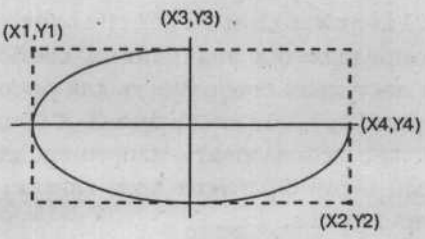

Рис. 8.9 ▾ Способ задания дуги

Сектор

Сектор задается тем же способом, что и дуга. Для рисования сектора необходимо использовать метод Pie:

```
<Компонент>.Canvas.Pie(x1,y1,x2,y2,x3,y3,x4,y4);
```

Параметры x1, y1, x2, y2 также задают прямоугольную область. Сектор вырезается против часовой стрелки от прямой, заданной точкой с координатами (x3, y3), к прямой, заданной точкой с координатами (x4, y4). Все вышесказанное иллюстрирует рис. 8.10.

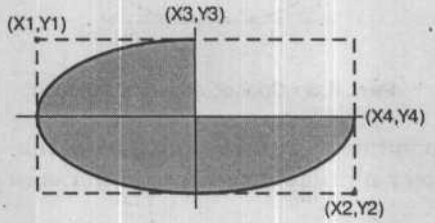

Рис. 8.10 ▼ Способ задания сектора

Точка

У поверхности Canvas, на которую осуществляется вывод графики, существует свойство Pixels, которое представляет собой двумерный массив типа TColor. Этот массив содержит информацию о цвете каждой точки графической поверхности. Используя свойство Pixels, можно задавать цвет для любой точки графической поверхности. Например, инструкция следующего вида

```
Form1.Canvas.Pixels[100,200]:=clBlue;
```

окрашивает точку (100, 200) в синий цвет.

Напомню, что размеры графической поверхности формы определяются значениями свойств ClientWidth и ClientHeight, в то время как общий размер компонента определяется значениями свойств Width и Height. Поэтому максимально доступная поверхность для рисования ограничивается прямоугольным контуром (0,0,ClientWidth-1,Clientheight-1).

Данное свойство можно использовать, например, для рисования графиков функций. Приведенный ниже фрагмент кода (листинг. 8.6) демонстрирует использование свойства Pixels.

Листинг 8.6 ▾ Рисование графика функции с помощью свойства `Pixels`

```
procedure TForm1.FormPaint(Sender: TObject);
var x0,y0:integer; // Координаты начала отсчета.
    i:integer;     // Счетчик цикла - 0..360 градусов.
begin
  x0:=50; y0:=100;
  for i:=1 to 360 do
    Canvas.Pixels[x0+i,y0-round(80*sin(PI*i/180))]:=clBlack;
end;
```

Результат использования данного кода приведен на рис. 8.11.

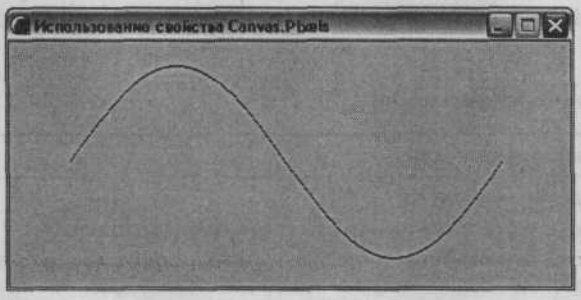

Рис. 8.11 ▾ Рисование графика функции y = sin(x) с помощью свойства `Pixels`

Вывод текста

Текст на графическую поверхность можно вставить с помощью метода `TextOut`:

```
<Компонент>.Canvas.TextOut(x,y,<Текст>);
```

Пара параметров (`x`, `y`) задает положение текста на экране, параметр `<Текст>` содержит строку текста, которую необходимо отобразить. В качестве примера использования метода `TextOut` можно привести фрагмент кода, приведенный в листинге 8.7 (процедура обработки события формы `OnPaint`).

Листинг 8.7 ▾ Использование метода `TextOut`

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Form1.Canvas do
    begin
      // Верхняя надпись - цвет фона совпадает с цветом кисти.
```

```
      Brush.Color:=clYellow;
      Font.Size:=18;
      Font.Style:=[fsItalic,fsBold];
      TextOut(10,30,'Работа с графикой в Win32');
      // Нижняя надпись - цвет фона совпадает с цветом формы.
      Brush.Color:=Form1.Color;
      TextOut(10,70,'Работа с графикой в Win32');
    end;
end;
```

В приведенном примере для вывода текста используется шрифт, заданный свойством `Font` соответствующего объекта `Canvas`. Ниже в табл. 8.8 приведены свойства объекта `Font`, задающие различные характеристики шрифта.

Таблица 8.8 ▼ Свойства объекта `TFont`

Свойство	Комментарий
`Name`	Определяет имя шрифта
`Size`	Размер шрифта. Задается в пунктах (1 пункт равен 1/72 дюйма)
`Style`	Свойство определяет стиль начертания шрифта. Стиль задается следующими константами: `fsBold` – полужирный шрифт, `fsItalic` – курсив, `fsUnderLine` – подчеркнутый, `fsStrikeOut` – зачеркнутый. Допускается любое сочетание констант
`Color`	Свойство определяет цвет символов с помощью константы типа `TColor`

Результат использования исходного кода, приведенного в листинге 8.7, представлен на рис. 8.12. Как видно из рисунка, область вывода текста закрашивается текущим цветом кисти. Поэтому если необходимо вывести текст без фонового цвета, то следует свойство кисти `Brush.Color` установить равным свойству `Form1.Color`.

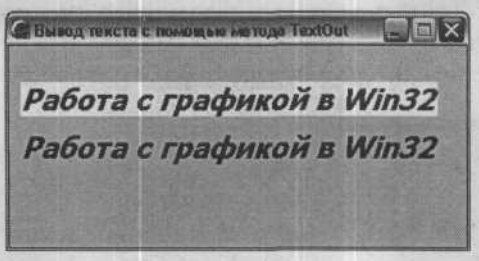

Рис. 8.12 ▼ Использование метода `TextOut` для вывода текста

Используем графические возможности для создания мультипликации

В главе 6 мы подробно рассмотрели основные вопросы, касающиеся мультипликации. В этой главе создадим аналогичную программу, позволяющую достичь того же эффекта. Основное внимание будет уделено отличиям новой программы от той, что была разработана в главе 6. Начнем разрабатывать новый вариант программы с применения уже знакомых нам битовых образов.

Использование битовых образов

В Win32 в отличие от .NET битовый образ создается в два этапа. Сначала необходимо применить метод `Create`. После этого следует применить метод `LoadFromFile`, которому в качестве параметра необходимо передать имя файла, хранящего изображение. Следующий фрагмент исходного кода демонстрирует создание битового образа:

```
var _image:TBitmap;
…
begin
…
_image:=TBitmap.Create;
_image.LoadFromFile('source.bmp');
…
end;
```

Для битового образа возможно задание прозрачного цвета. Для этого сначала необходимо установить значение свойства `Transparent` в `True`, а затем указать цвет, который будет являться прозрачным. Соответственно, все точки, имеющие данный цвет, отображаться не будут. Вышесказанное поясняет приведенный ниже код:

```
_image.Transparent:=True;
_image.TransparentColor:=_image.Canvas.Pixels[1,1];
```

Таким образом, битовый образ сформирован. Далее картинку, хранимую в памяти компьютера, можно вывести на поверхность формы методом `Draw`. В качестве параметров нужно указать битовый образ, который собираемся отображать, и координаты точки поверхности, с которой будет осуществляться вывод. Например, следующий оператор позволяет вывести битовый образ в левом верхнем углу формы:

```
Form1.Canvas.Draw(0,0,_image);
```

Следующая программа (листинг 8.8) демонстрирует создание и отображение битовых образов.

Листинг 8.8 ▼ Работа с битовыми образами

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  sky,parashut:TBitmap;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  // Загружаем фон.
  sky:=TBitmap.Create;
  sky.LoadFromFile('sky.bmp');
  // Изменяем размер окна под размер фоновой картинки.
  Width:=sky.Width;
  Height:=sky.Height;
  // Загружаем изображение парашютиста.
  parashut:=TBitmap.Create;
  parashut.LoadFromFile('parashut.bmp');
end;

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

```
  // Отображаем фон (небо).
  Form1.Canvas.Draw(0,0,sky);
  // Левый парашютист.
  Form1.Canvas.Draw(100,80,parashut);
  // Правый парашютист.
  // Устанавливаем  'прозрачный' цвет
  // (цвет левой нижней точки).
  parashut.Transparent:=True;
  parashut.TransparentColor:=parashut.Canvas.Pixels[1,1];
  Form1.Canvas.Draw(300,80,parashut);
end;

end.
```

В результате окно нашей программы будет выглядеть так, как показано на рис. 8.13.

Рис. 8.13 ▾ Использование битовых образов и установки прозрачного цвета

Мультипликация с помощью битовых образов

После того как мы научились отображать битовые образы в Win32, самое время перейти к мультипликации. Для этого нам потребуется последовательно отображать битовые образы в разных местах экрана, тем самым создавая иллюзию движения.

Немного изменим текст программы по аналогии с известным нам подходом, рассмотренным в главе 6. Приведенный ниже в листинге 8.9 исходный код демонстрирует данный прием мультипликации.

Листинг 8.9 ▼ Мультипликация с помощью битовых образов

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls;

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  sky,parashut:TBitmap;
  rect,drect:TRect;     // Структуры для хранения прямоугольных
                        // областей.
  d:TBitmap;            // Буфер.
  x,y:integer;     // Текущие координаты парашютиста (левого
                   // верхнего угла).
  _width,_height:integer;// Длина и высота копируемого фрагмента.

implementation

{$R *.dfm}

procedure TForm1.FormActivate(Sender: TObject);
begin
  // Создаем и отображаем фон (небо).
  sky:=TBitmap.Create;
  sky.LoadFromFile('sky.bmp');
  Form1.ClientWidth:=sky.Width;
  Form1.ClientHeight:=sky.Height;
  Form1.Canvas.Draw(0,0,sky);
  // Создаем парашютиста.
  parashut:=TBitmap.Create;
```

```
  parashut.Create.LoadFromFile('parashut.bmp');
  parashut.Transparent:=True;
  parashut.TransparentColor:=parashut.Canvas.Pixels[1,1];
  // Настраиваем буфер.
  d:=TBitMap.Create;
  _width:=parashut.Width;
  _height:=parashut.Height;
  // Настраиваем буфер, где будет храниться картинка.
  d.Width:=_width;
  d.Height:=_height;
  d.Palette:=sky.Palette;
  d.Canvas.CopyMode:=cmSrcCopy;
  rect:=Bounds(0,0,_width,_height);
  // Начальное положение парашютиста.
  x:=200;
  y:=20;
  // Сохраненяем область фона, на которую накладывается картинка.
  drect:=Bounds(x,y,_width,_height);
  d.Canvas.CopyRect(rect,sky.Canvas,drect);
end;

procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  sky.Free;
  parashut.Free;
  d.Free;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  // Восстанавливаем фон из буфера (удаляем рисунок).
  Form1.Canvas.Draw(x,y,d);
  y:=y+1;
  if y=150 then Timer1.Enabled:=False;
  // Определим сохраняемую часть фона.
  drect:=Bounds(x,y,_width,_height);
  // Сохраним ее в буфер.
  d.Canvas.CopyRect(rect,sky.Canvas,drect);
  // Выведем рисунок в указанной области.
  Form1.Canvas.Draw(x,y,parashut);
end;

end.
```

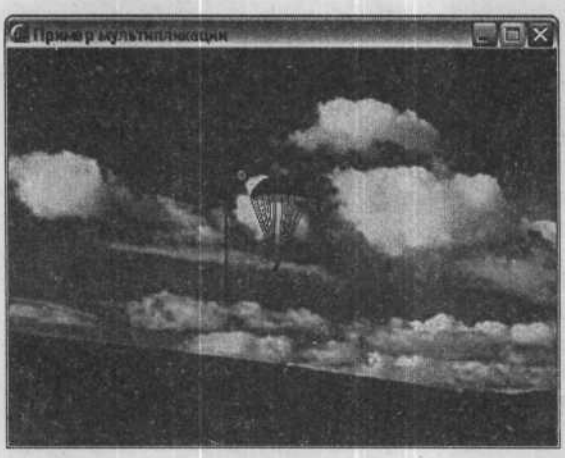

Рис. 8.14 ▼ Движение битового образа

Программа работает следующим образом. Сначала объект выводится на созданную нами графическую поверхность d, через некоторое время стирается и выводится снова, но уже на некотором расстоянии dy от первоначального положения. Таким образом, парашютист «движется» вертикально (рис. 8.14).

Заключение

Подведем некоторые итоги. Мы закончили изучение основ работы в среде Borland Delphi 2005. Вы узнали, что представляет собой среда разработки Borland Delphi 2005 в целом, как создаются программы, а также немного познакомились с языком программирования Delphi.

По-моему, на этом пока можно остановиться. Согласен, мы рассмотрели далеко не все, но книга задумывалась как средство, которое позволит начинающему пользователю познакомиться с этим мощным программным продуктом, а не дать ответы на все вопросы о среде разработки Borland Delphi 2005. Конечно, если взяться за подробное изложение всех возможностей этого продукта, то будет невозможно ограничиться таким объемом книги, да и тогда эта книга потеряет изначально заложенный в нее смысл.

Я очень надеюсь, что оправдал ваши ожидания, и книга действительно стала вашим хорошим помощником и верным спутником на пути освоения среды программирования Borland Delphi 2005 и языка Delphi.

Если вы прочитали ее полностью, а главное – попробовали поэкспериментировать с приведенными примерами, а также сделать что-нибудь своими руками, то примите мои поздравления – вы проделали очень большую работу. Прежде всего – для себя. Теперь вы уже не начинающий программист, и если хотите осваивать программирование дальше, то вам вполне по силам читать уже совершенно другие книги.

Приложение

Примеры программ

Все рассмотренные в книге примеры программ доступны для бесплатной загрузки с сайта издательства www.ntpress.ru/download/delphi.rar. Каждый загружаемый проект после распаковки представляет собой отдельный каталог, в котором находятся все необходимые файлы – исходные коды проекта, исполняемый файл и вспомогательные файлы (там, где это необходимо).

Ниже в табл. П1.1 приведен полный перечень проектов, а также их краткое описание.

Таблица П1.1 ▾ Перечень примеров программ, рассматриваемых в книге

Проект	Глава книги	Комментарий
Example_01	3	Программа вычисления корней квадратного уравнения, написанная для платформы .NET
Example_02	3	Программа вычисления корней квадратного уравнения, написанная для платформы Win32
Example_03	5	Пример программы, демонстрирующей основные свойства компонента `Label`
Example_04	5	Пример программы, демонстрирующей основные особенности компонента `TextBox`. В программе реализована фильтрация вводимых символов в поле компонента
Example_05	5	Пример программы, демонстрирующей основные особенности компонента `Panel`
Example_06	5	Пример программы, демонстрирующей основные особенности компонента `CheckBox`. Программа позволяет вычислить стоимость заказа
Example_07	5	Пример программы, демонстрирующей основные особенности компонента `RadioButton`. Программа позволяет задавать опции туристической поездки
Example_08	5	Пример программы, демонстрирующей основные особенности компонента `PictureBox`. Программа позволяет просматривать файлы изображений из каталога
Example_09	5	Пример программы, демонстрирующей основные особенности компонента `PictureBox`. Программа позволяет задавать основные режимы отображения картинки в поле компонента
Example_10	5	Пример программы, демонстрирующей основные особенности компонента `StatusBar`
Example_11	5	Пример программы, демонстрирующей основные особенности компонента `StatusBar`. Программа представляет собой простейший секундомер
Example_12	5	Пример программы, демонстрирующей основные особенности компонента `ComboBox`. Программа представляет собой вариант тестового опроса

Таблица П1.1 ▼ Перечень примеров программ, рассматриваемых в книге (продолжение)

Проект	Глава книги	Комментарий
Example_13	5	Пример программы, демонстрирующей основные особенности компонента `ToolBar`. Программа представляет собой простейший текстовый редактор
Example_14	5	Пример программы, демонстрирующей основные особенности компонента `ProgressBar`. Программа выполняет копирование файлов
Example_15	5	Пример программы, демонстрирующей основные особенности компонента `MainMenu`. Программа представляет собой простейший текстовый редактор с главным меню
Example_16	5	Пример программы, демонстрирующей основные особенности компонента `ContextMenu`
Example_17	5	Пример программы, демонстрирующей основные особенности компонента `OpenFileDialog`
Example_18	5	Пример программы, демонстрирующей основные особенности компонента `SaveFileDialog`
Example_19	6	Пример программы, демонстрирующей основные возможности графики. Рисует на поверхности окна простейшие графические примитивы, а также отображает рисунок
Example_20	6	Пример программы, демонстрирующей создание и использование карандаша (объекта `Pen`)
Example_21	6	Пример программы, демонстрирующей использование штриховой кисти
Example_22	6	Пример программы, демонстрирующей использование текстурной кисти
Example_23	6	Пример программы, демонстрирующей использование градиентной кисти
Example_24	6	Пример программы, демонстрирующей использование расширенных возможностей метода `DrawString`
Example_25	6	Пример программы, демонстрирующей создание и использование битовых образов
Example_26	6	Пример программы, демонстрирующей создание мультипликации с помощью битовых образов
Example_27	6	Пример программы, демонстрирующей использование GIF-анимации
Example_28	7	Пример программы, демонстрирующей основные особенности компонента `TLabel`
Example_29	7	Пример программы, демонстрирующей основные особенности компонента `TPanel`
Example_30	7	Пример программы, демонстрирующей основные особенности компонента `TRadioButton` и `TRadioGroup`
Example_31	7	Пример программы, демонстрирующей вариант использования компонента `TComboBox`
Example_32	7	Пример программы, демонстрирующей вариант использования компонента `TListBox`
Example_33	7	Пример программы, демонстрирующей различные режимы отображения картинок с использованием компонента `TImage`
Example_34	7	Пример программы, демонстрирующей особенности использования компонента `TUpdownw`

Таблица П1.1 ▾ Перечень примеров программ, рассматриваемых в книге (окончание)

Проект	Глава книги	Комментарий
Example_35	7	Пример программы, демонстрирующей особенности использования компонента `TStatusBar`
Example_36	7	Пример программы, демонстрирующей особенности использования компонента `TTimer`. Программа представляет собой простейший секундомер
Example_37	7	Пример программы, демонстрирующей особенности использования компонента `TPopupMenu`
Example_38	7	Пример программы, демонстрирующей особенности использования компонента `TOpenDialog`
Example_39	7	Пример программы, демонстрирующей особенности использования компонента `TSaveDialog`
Example_40	8	Пример программы, демонстрирующей особенности использования объекта «кисть» (`TBrush`)
Example_41	8	Пример программы, демонстрирующей особенности использования методов `LineTo` и `PolyLine`. Программа представляет собой пример рисования координатной сетки с графиком
Example_42	8	Пример программы, демонстрирующей особенности использования методов рисования прямоугольников
Example_43	8	Пример программы, демонстрирующей особенности метода рисования многоугольника. Программа рисует правильный восьмиугольник
Example_44	8	Пример программы, демонстрирующей особенности использования свойства графической поверхности `Canvas.Pixels`. Программа рисует график функции `y=sin(x)`
Example_45	8	Пример программы, демонстрирующей особенности использования метода вывода текста (метод `TextOut`)
Example_46	8	Пример программы, демонстрирующей создание и использование битовых образов
Example_47	8	Пример программы, демонстрирующей создание мультипликации с использованием битовых образов
Example_48	8	Пример программы, демонстрирующей основные графические возможности методов в Win32
Example_49	4	Пример программы, демонстрирующей создание и использование простейшего класса
Example_50	4	Пример программы, демонстрирующей создание и использование простейшего класса с применением свойств (`properties`), пояснение принципа инкапсуляции
Example_51	4	Пример программы, демонстрирующей создание и использование простейших класса-родителя и класса-потомка, пояснение принципа наследования
Example_52	4	Пример программы, демонстрирующей создание и использование простейших классов, использующих одноименные методы. Пояснение принципа полиморфизма