МИХАИЛ АБРАМЯН

Visual C#

НА ПРИМЕРАХ

РАЗРАБОТКА ПРИЛОЖЕНИЙ
В MICROSOFT VISUAL C# 2005/2008

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ КЛАССОВ
WINDOWS FORMS

РЕАЛИЗАЦИЯ УДОБНОГО
И НАДЕЖНОГО ИНТЕРФЕЙСА

ОБСУЖДЕНИЕ ТИПИЧНЫХ ОШИБОК
И СПОСОБОВ ИХ ИСПРАВЛЕНИЯ

+CD

bhv®

Михаил Абрамян

Visual C#

НА ПРИМЕРАХ

Санкт-Петербург

«БХВ-Петербург»

2008

УДК 681.3.068+800.92Visual C#
ББК 32.973.26-018.1
А16

Абрамян М. Э.

А16 Visual C# на примерах. — СПб.: БХВ-Петербург, 2008. — 496 с.: ил. + CD-ROM

ISBN 978-5-9775-0266-5

Книга содержит подробное описание 32 проектов, демонстрирующих различные аспекты создания Windows-приложений для платформы .NET Framework в среде Microsoft Visual C# 2005/2008. Рассматриваются оптимальные приемы разработки программ, управляемых событиями, механизм обработки исключений, особенности консольных и MDI-приложений. Детально описываются основные компоненты библиотеки Windows Forms и классы, входящие в графическую библиотеку GDI+. Демонстрируются приемы работы с клавиатурой и мышью, а также дополнительные возможности .NET-приложений, в том числе реализация режима перетаскивания drag & drop, работа с реестром Windows и др. На компакт-диске содержатся исходные тексты проектов, описанных в книге.

Для программистов

УДК 681.3.068+800.92Visual C#
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.06.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 39,99.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

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

ISBN 978-5-9775-0266-5

Оглавление

Памяти моего учителя профессора
Игоря Борисовича Симоненко (1935—2008)

Предисловие

Книга, предлагаемая вашему вниманию, содержит подробные указания по разработке 32 примеров Windows-приложений для платформы .NET Framework. При создании приложений используется язык Visual C#. Предполагается, что в качестве интегрированной среды используется среда Microsoft Visual Studio .NET 2005 или 2008.

Каждый из примеров книги посвящен определенной теме, указанной в его названии, однако при описании процесса разработки проектов и в сопутствующих комментариях приводятся дополнительные сведения, что позволяет охватить очень широкий диапазон вопросов, связанных с созданием Windows-приложений. При этом особое внимание уделяется оптимальным приемам разработки приложений, управляемых событиями, и эффективному применению компонентов библиотеки Microsoft Windows Forms. Подробно обсуждаются средства, обеспечивающие удобный и надежный диалог программы с пользователем. Кроме того, рассматриваются типичные ошибки, возникающие при использовании различных классов библиотеки Windows Forms, и указываются способы их исправления.

Книга ориентирована на читателя, владеющего основами программирования. Желательно также знакомство с языком C# (на уровне знания системы базовых типов и управляющих операторов). Для ознакомления с языком C# можно воспользоваться, например, книгой [4], являющейся, к тому же, мастерски написанным учебником по программированию. Глубокого знания *объектной модели* платформы .NET не требуется. Основные термины, связанные с объектной моделью .NET, кратко объясняются в словаре, включенном в книгу в качестве *приложения 1*.

В версии .NET 2.0 библиотека Microsoft Windows Forms подверглась существенной переработке, поэтому многие возможности, описанные в книге, будут

недоступны для более ранних версий .NET (в частности, для версии .NET 1.1 и ориентированной на нее среды Microsoft Visual Studio .NET 2003). В версиях .NET 3.0 и 3.5 библиотека Microsoft Windows Forms практически не изменилась. Для того чтобы сохранить совместимость с широко распространенной версией .NET 2.0 и средой Microsoft Visual Studio .NET 2005, в книге не используются возможности, добавленные в язык Visual C#, начиная с версии 3.0. Следует отметить, что эти возможности (в частности, технология LINQ — Language-Integrated Query — технология запросов, интегрированных в язык программирования) в основном ориентированы на приложения, связанные с обработкой баз данных, которые в книге не рассматриваются.

Кратко опишем содержимое книги.

Глава 1 знакомит читателя с основными приемами работы в среде Microsoft Visual Studio .NET. Пример DISKINFO *(см. главу 2)* посвящен особенностям разработки консольных .NET-приложений. Пример EXCEP *(см. главу 3)*, также являющийся консольным приложением, демонстрирует различные аспекты обработки исключительных ситуаций.

Далее следует группа из 10 примеров, посвященных основам разработки графических .NET-приложений. В примере EVENTS *(см. главу 4)* рассматриваются вопросы, связанные с событиями и их обработкой. Пример FORMS *(см. главу 5)* посвящен формам и их взаимодействию в рамках одного приложения. Пример CALC *(см. главу 6)* знакомит с возможностью совместного использования обработчика для нескольких событий, а также описывает различные приемы работы с клавиатурой. Пример CURSORS *(см. главу 7)* показывает, как использовать в приложениях курсоры и значки (пиктограммы). Пример TEXTBOXES *(см. главу 8)* знакомит с различными возможностями полей ввода, а также описывает средства, связанные с проверкой введенных данных и выводом сообщений об ошибках. Пример COLORS *(см. главу 9)* знакомит с типами, предназначенными для работы с цветом, и с компонентами, обеспечивающими прокрутку данных. Пример MOUSE *(см. главу 10)* посвящен событиям, связанным с манипулятором "мышь". В примере ZOO *(см. главу 11)* описывается, как реализовать режим перетаскивания drag & drop. Пример CLOCK *(см. главу 12)* знакомит с типами, связанными с датами и временем. Наконец, пример IMGVIEW *(см. главу 13)* посвящен компонентам, обеспечивающим навигацию по файловой системе; кроме того, в нем показано, как использовать реестр Windows для хранения информации о текущих настройках приложения.

Следующая группа из 10 примеров посвящена реализации двух полнофункциональных редакторов: графического и текстового. Разработка графического редактора (с применением средств графической библиотеки GDI+) проводится в серии примеров PNGEDIT. В примере PNGEDIT1 *(см. главу 14)* рассматрива-

ются основные действия с графическими файлами и приводится простейшая реализация рисования тонким одноцветным пером. Пример PNGEDIT2 *(см. главу 15)* показывает, как реализовать в редакторе рисование цветным пером произвольной толщины и наглядное рисование прямых линий. В примере PNGEDIT3 *(см. главу 16)* редактор дополняется возможностью рисования замкнутых фигур (прямоугольников и эллипсов) в обычном режиме и в режиме прозрачности, т. е. без заливки внутренности фигуры фоновым цветом. В примере PNGEDIT4 *(см. главу 17)* графический редактор снабжается некоторыми полезными дополнительными возможностями, среди которых следует отметить отмену последней графической операции, настройку цветов с помощью инструмента "пипетка", добавление к изображению текста и настройку стиля линии.

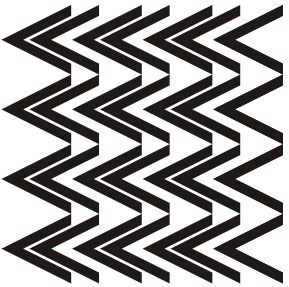
Разработка текстового редактора проводится в серии примеров TXTEDIT. В примере TXTEDIT1 *(см. главу 18)* описывается создание простейшего меню приложения и реализуются основные действия с текстовыми файлами. В примере TXTEDIT2 *(см. главу 19)* рассматриваются дополнительные возможности, связанные с меню (команды-флажки и команды-переключатели). В примере TXTEDIT3 *(см. главу 20)* реализуются стандартные команды редактирования и рассматривается работа с контекстным меню. Пример TXTEDIT4 *(см. главу 21)* описывает возможности панели инструментов, а пример TXTEDIT5 *(см. главу 22)* — статусной панели. Наконец, в примере TXTEDIT6 *(см. главу 23)* описываются действия по преобразованию созданного текстового редактора в форматирующий, т. е. позволяющий по-разному форматировать различные фрагменты текста и сохранять форматные настройки вместе с текстом. При этом подробно рассматриваются возможности компонента `RichTextBox`.

Завершающая группа из 10 примеров посвящена углубленному изучению некоторых важных компонентов библиотеки Microsoft Windows Forms. В примере CHKBOXES *(см. главу 24)* описываются различные возможности компонентов-флажков, а также групп флажков; в серии из двух примеров LISTBOX1–LISTBOX2 *(см. главы 25 и 26)* рассматриваются выпадающие и обычные списки. Пример JPEGVIEW *(см. главу 27)* посвящен разработке приложения с многодокументным интерфейсом (MDI-приложения). В следующих пяти примерах основную роль играет компонент-таблица `DataGridView`. Пример FONTVIEW *(см. главу 28)* посвящен приемам работы со шрифтами и их семействами; в нем таблица используется для отображения символов различных шрифтов. В примере ICONVIEW *(см. главу 29)* реализуется возможность чтения значков из exe- и dll-файлов; в нем таблица `DataGridView` используется для отображения графической информации. В этом примере также показывается, как импортировать стандартные функции Win32 API в .NET-приложение, и приводится пример создания объектной оболочки для импортированной функции. Пример

TRIGFUNC *(см. главу 30)* показывает, как создать окно-заставку с особыми свойствами; кроме того, в нем содержатся дополнительные сведения о настройке свойств таблицы `DataGridView`. Следующие два примера STUD1–STUD2 *(см. главы 31 и 32)* являются наиболее сложными. Эти примеры посвящены средствам .NET, предназначенным для обработки наборов данных, включая их привязку (binding) к компонентам формы. В них также рассматривается XML-сериализация наборов данных, работа с подчиненными наборами данных и способы сортировки и поиска данных, основанные на применении функций сравнения и тестовых функций. Последний пример HTOWERS *(см. главу 33)* представляет собой компьютерную реализацию известной логической задачи "Ханойские башни". В этом примере рассматриваются особенности, связанные с созданием компонентов в ходе выполнения программы. Помимо обычного режима, в котором пользователь может самостоятельно решать задачу (используя перетаскивание drag & drop), приложение имеет демонстрационный режим, показывающий правильную последовательность действий для решения задачи. В заключительном разделе примера HTOWERS описывается, как задать идентификационные данные для разработанного приложения.

На прилагаемом компакт-диске содержатся все проекты, описанные в книге, причем для каждого проекта приводится несколько вариантов, соответствующих различным этапам его разработки.

Книга написана при поддержке внутреннего гранта К–07–Т–87 Южного федерального университета.



Часть I

Глава 1

Разработка программ в среде Microsoft Visual Studio .NET

В книге описываются приемы работы в двух вариантах интегрированных сред, созданных корпорацией Microsoft и ориентированных на платформу .NET Framework. Первый вариант — свободно распространяемая бесплатная система Microsoft Visual C# Express Edition, второй вариант — коммерческая система Microsoft Visual Studio, позволяющая разрабатывать приложения на различных языках платформы .NET. В среде Microsoft Visual Studio предусмотрены средства для разработки существенно большего числа типов приложений, однако в отношении стандартных Windows-приложений возможности данной среды по существу совпадают с возможностями ее экспресс-варианта для языка C#.

В настоящее время наиболее распространены две версии систем Microsoft Visual C# Express Edition и Microsoft Visual Studio: версия 2005, ориентированная на платформу .NET 2.0, и новая версия 2008, ориентированная на платформу .NET 3.5. В книге рассматриваются возможности, имеющиеся в обеих версиях.

В дальнейшем под средой Visual C# мы будем подразумевать как среду Microsoft Visual C# 2005/2008, входящую в состав системы Microsoft Visual Studio 2005/2008, так и ее экспресс-вариант.

1.1. Создание, сохранение и открытие проекта

При запуске среды Visual C# в нее автоматически загружается стартовая страница (**Start Page**), позволяющая быстро загрузить один из ранее разрабатывавшихся проектов (список **Recent Projects**), открыть какой-либо другой существующий проект (пункт **Open: Project...**) и создать новый проект (пункт **Create: Project...**). Впрочем, все отмеченные возможности доступны также из меню среды Visual C#:

- **File | New | Project...** или комбинация клавиш <Ctrl>+<Shift>+<N> — создание нового проекта (в экспресс-варианте данная команда является командой меню второго уровня: **File | New Project...**);
- **File | Open | Project/Solution...** или комбинация клавиш <Ctrl>+<Shift>+ +<O> — открытие существующего проекта (в экспресс-варианте команда имеет вид **File | Open Project...**);
- **File | Recent Projects** — загрузка одного из недавно разрабатывавшихся проектов.

Среда Visual C# организована таким образом, что в ней нельзя создать "отдельно взятый" проект. Каждый проект должен содержаться в особой сущности, называемой solution ("решение"), которую можно охарактеризовать как *группу взаимосвязанных проектов* (по этой причине в данной книге термин solution мы будем переводить как "группа проектов"). В каждый момент времени в среду Visual C# можно загрузить только одну группу проектов; загрузка другой группы приводит к автоматическому закрытию предыдущей.

При создании нового проекта на экране возникает диалоговое окно, в котором надо выбрать тип проекта и его имя. В качестве типа для всех проектов, рассматриваемых в книге (кроме двух первых проектов DISKINFO и EXCEP), следует выбирать **Windows Application**; в качестве имени рекомендуется указывать имя примера, приведенное в заголовке соответствующей главы книги, например, EVENTS *(см. главу 4)*. Заметим, что имя проекта может содержать не только цифры и латинские буквы, но и другие символы, допустимые в именах файлов, в том числе пробелы и русские буквы, хотя этой возможностью не следует злоупотреблять.

В среде Visual Studio при создании нового проекта необходимо сразу указать каталог для его сохранения (поле **Location**). На размещение проекта также влияет информация, связанная с группой проектов (solution), в которую будет помещен создаваемый проект, в частности, флажок **Create directory for solution** (Создать каталог для группы проектов). Если в группу будет входить единственный проект, то следует снять данный флажок; это приведет к тому, что созданная группа проектов будет иметь то же имя, что и созданный проект, а в каталоге, указанном в поле **Location**, будет создан каталог с именем проекта, в котором будут размещаться все файлы, связанные с проектом и его группой.

При установленном флажке **Create directory for solution** можно указать имя группы проектов, отличное от имени самого проекта. В этой ситуации создается более сложная иерархия каталогов: в каталоге, указанном в поле **Location**, будет создан каталог с именем группы проектов, а в нем — каталог с именем проекта.

Отметим еще одну возможность среды Visual Studio (которой мы, однако, пользоваться не будем): создаваемый проект можно добавить к текущей группе проектов, т. е. группе, ранее загруженной в среду Visual Studio.

В экспресс-варианте Visual C# при создании нового проекта не требуется указывать его расположение; вся необходимая информация запрашивается при первом *сохранении* созданного проекта, поэтому рекомендуется выполнить это сохранение сразу после создания проекта.

Для сохранения всех изменений, внесенных в текущий проект (а точнее, в текущую группу проектов), в Visual C# предусмотрена команда **File | Save All**, а также клавиатурная комбинация <Ctrl>+<Shift>+<S>.

При открытии существующего проекта на экране появляется диалоговое окно **Open**, в котором можно выбрать либо файл с расширением sln (содержащий информацию о группе проектов с указанным именем), либо файл с расширением csproj (содержащий информацию о проекте с указанным именем). Однако даже при выборе отдельного проекта загружается группа проектов, в которой он содержится.

Некоторые примеры, описанные в книге, предполагают не создание нового проекта "с нуля", а модификацию уже имеющегося проекта. Имена таких примеров снабжены порядковыми номерами, начиная с номера 2 (например, PNGEDIT2—PNGEDIT4). Перед началом выполнения подобных примеров следует скопировать в новый каталог все файлы проекта из примера с предыдущим номером, после чего загрузить полученную копию проекта в среду Visual C# и начать ее модификацию.

Действия по модификации проекта опишем на примере преобразования проекта PNGEDIT1 в проект PNGEDIT2.

Вначале следует изменить имя проекта и имя связанной с ним группы проектов. Это проще всего выполнить с помощью окна **Solution Explorer**, обычно расположенного в правой части экрана (рис. 1.1): достаточно выделить в окне **Solution Explorer** строку, соответствующую проекту (на рис. 1.1 эта строка является выделенной) или группе проектов (данная строка начинается со слова Solution), нажать клавишу <F2>, ввести новое имя и нажать клавишу <Enter>.

Следует также изменить имя сборки (assembly name), т. е. имя результирующего exe-файла. Для этого надо выполнить команду **Project | *<Имя проекта>* Properties** (в нашем случае **Project | PNGEDIT2 Properties**), перейти в появившейся вкладке с именем проекта в раздел настроек **Application** и указать новое имя в поле **Assembly name** (рис. 1.2; для версии 2008 вид раздела **Application** несколько отличается от приведенного на рисунке). Название пространства

имен по умолчанию (поле **Default namespace**) изменять не следует, поскольку прежнее название используется во всех имеющихся cs-файлах проекта.

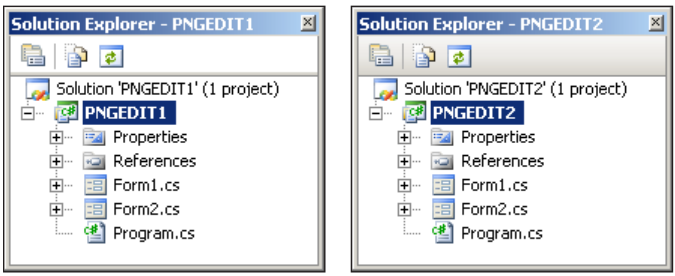


Рис. 1.1. Вид окна **Solution Explorer** до изменения имени проекта *(слева)* и после его изменения *(справа)*

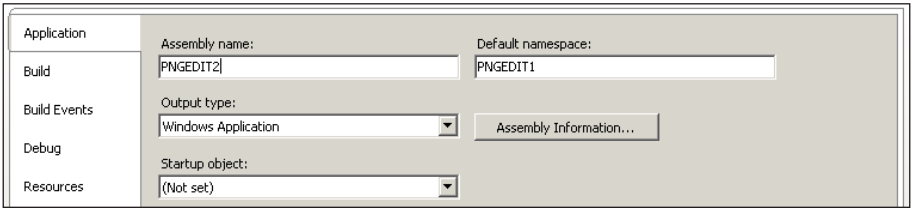


Рис. 1.2. Вид верхней части окна со свойствами проекта для версии Visual C# 2005

1.2. Добавление к проекту новой формы и размещение в форме нового компонента

При создании нового проекта типа Windows Application связанная с ним главная форма (класс с именем `Form1`) создается автоматически и сразу загружается в среду Visual C#. С формой `Form1` связывается набор файлов, основными из которых являются файлы Form1.cs и Form1.Designer.cs. В этих файлах содержится описание данной формы на языке C#, причем второй из них содержит ту часть описания, которая генерируется автоматически в ответ на различные действия разработчика, связанные с визуальным проектированием (собственный код разработчик обычно размещает в файле Form1.cs).

Если в проекте требуется использовать дополнительные формы *(см., например, главу 5)*, то их проще всего добавить в проект с помощью команды меню **Project | Add Windows Form...**. При выполнении этой команды возникает диалоговое окно, в котором надо указать имя добавляемой формы (которое одновременно будет и начальной частью имен файлов, содержащих ее описание).

В книге всегда используются имена форм, предлагаемые системой по умолчанию (`Form1`, `Form2` и т. д.).

Любая созданная форма отображается в редакторе Visual C# в *режиме дизайна* (на экране появляется изображение формы и ее содержимого); соответствующая вкладка редактора заканчивается текстом **[Design]**. Например, для главной формы `Form1` вкладка содержит текст **Form1.cs [Design]**. Для перехода к содержимому соответствующего cs-файла достаточно нажать клавишу <F7>; при этом в редакторе появится новая вкладка с текстом данного файла (ярлычок этой вкладки будет содержать имя файла, например, **Form1.cs**). Для обратного переключения с текста на изображение формы можно использовать комбинацию клавиш <Shift>+<F7>. Переключаться между изображением формы и текстом кода можно также с помощью *контекстного меню*: при щелчке правой кнопкой мыши на изображении формы первый пункт появляющегося меню имеет имя **View Code** и позволяет перейти к тексту cs-файла, а при вызове контекстного меню для текста cs-файла первый пункт меню имеет имя **View Designer** и позволяет вернуться в режим дизайна.

Другим удобным способом переключения между различными окнами среды Visual C# является использование комбинации клавиш <Ctrl>+<Tab>: после ее нажатия на экране появляется вспомогательное окно со списком всех загруженных файлов и форм (Active Files), а также всех открытых вспомогательных окон (Active Tool Windows). На рис. 1.3 приведен примерный вид вспомогательного окна для версии 2005 (в версии 2008 окно имеет незначительные отличия). После появления вспомогательного окна надо, не отпуская клавиши <Ctrl>, выделить имя файла или окна, которое надо активизировать (для выбора можно использовать клавиши со стрелками и клавишу <Tab>), а затем отпустить клавишу <Ctrl>.

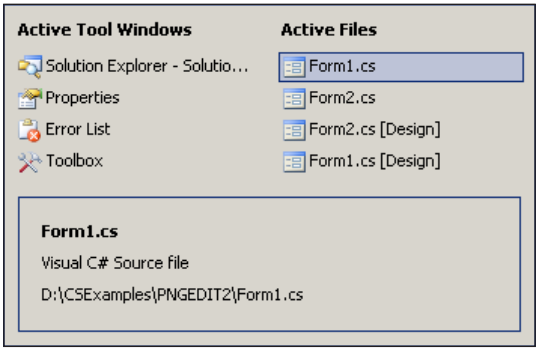


Рис. 1.3. Вспомогательное окно для выбора одного из загруженных файлов или окон среды Visual C#

Любую вкладку в редакторе можно закрыть. Для этого достаточно сделать ее активной и нажать комбинацию клавиш <Ctrl>+<F4> или вызвать контекстное меню вкладки, щелкнув правой кнопкой мыши на ее ярлычке, и выбрать пункт **Close**.

Если после загрузки существующего проекта в редактор не загрузилась требуемая форма, достаточно выполнить двойной щелчок мышью на имени этой формы в окне **Solution Explorer** (см. рис. 1.1). Можно также щелкнуть на имени этой формы *правой* кнопкой мыши и выбрать пункт **Open** из появившегося контекстного меню. Данное меню также содержит пункт **View Code**, позволяющий загрузить в редактор текст cs-файла, и пункт **View Designer**, позволяющий загрузить в редактор форму в режиме дизайна. При выборе одной из форм в окне проекта **Solution Explorer** дополнительно появляются кнопки быстрого доступа  , первая из которых позволяет загрузить текст файла, а вторая — отобразить форму в режиме дизайна.

Для размещения в форме нового компонента следует воспользоваться *окном компонентов* **Toolbox** (отобразить это окно на экране можно либо командой меню **View | Toolbox**, либо комбинацией клавиш <Ctrl>+<Alt>+<X>, либо кнопкой ). Это окно обычно отображается в левой части экрана; на рис. 1.4 приведен вид верхней части окна **Toolbox** при выборе в нем группы **All Windows Forms**. Заметим, что компоненты отображаются в окне **Toolbox** только в том случае, если редактор находится в режиме дизайна. Надо выбрать в этом окне группу, содержащую нужный компонент (щелкнув мышью на имени этой группы), затем щелкнуть на изображении нужного компонента, выделив его (на рис. 1.4 выделен компонент `Button`), и, наконец, щелкнуть в том месте формы, где предполагается разместить выбранный компонент.

Для того чтобы выбрать компонент в окне **Toolbox**, необязательно знать, в какой группе он содержится, поскольку в этом окне имеется группа **All Windows Forms**, в которой указаны все имеющиеся компоненты в алфавитном порядке (рис. 1.4).

Если в окне **Toolbox** отсутствует нужный компонент, то можно попытаться загрузить его в это окно. Действия, которые при этом надо выполнить, описываются в *разд. 13.1*.

Для быстрого размещения в форме нескольких компонентов одного типа следует после выделения требуемого компонента в окне **Toolbox** выполнить щелчок мышью на форме несколько раз, держа нажатой клавишу <Ctrl>. Если в окне **Toolbox** требуется отменить выбор компонента, не размещая его в форме, то достаточно выделить в этом окне элемент **Pointer** (он располагается первым в любой группе компонентов; рядом с ним изображен курсор в виде стрелки — см. рис. 1.4).

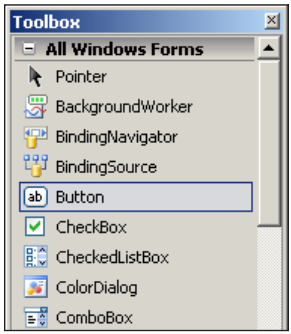


Рис. 1.4. Вид верхней части окна **Toolbox**

Если компонент требуется разместить не в форме, а в другом компоненте (например, в компоненте `Panel`), то при размещении компонента необходимо выполнить щелчок мыши в области компонента-приемника. В качестве компонентов-приемников могут использоваться только особые компоненты, называемые *компонентами-контейнерами* (сама форма также является компонентом-контейнером). Компонент-контейнер (форма, панель и т. д.), содержащий другие компоненты, называется *родительским* (parent) по отношению к размещенным в нем *дочерним* компонентам.

При описании этапов разработки проекта мы всегда будем указывать, в каком компоненте-контейнере следует разместить каждый компонент.

После добавления компонента в форму он автоматически получает *имя*, которое сохраняется в его свойстве `Name`. По умолчанию имя любого компонента начинается со строчной буквы и состоит из имени типа компонента и порядкового номера. Так, первая размещенная в форме кнопка (компонент типа `Button`) получит имя `button1`, а вторая кнопка — имя `button2`. Формы имеют имена, начинающиеся с прописной буквы (`Form1`, `Form2` и т. д.); это связано с тем, что данные имена являются именами новых *классов* — потомков базового класса `Form`. Имя, присвоенное по умолчанию, всегда можно заменить на другое имя, позволяющее, например, уточнить назначение того или иного компонента (например, кнопку `button1`, содержащую текст **OK**, можно назвать `btnOK`). Однако в проектах, описываемых в данной книге, почти всегда используются имена компонентов, предлагаемые системой Visual C# по умолчанию: это позволяет уменьшить количество действий по настройке свойств компонентов и упрощает ориентировку в текстах программ. "Значимые" имена используются только для пунктов меню, кнопок быстрого доступа и разделов статусной панели *(см. главы 18, 21, 22)*. Следует заметить, однако, что при разработке больших проектов целесообразно использовать значимые имена для всех компонентов.

При описании действий по добавлению компонента в форму почти никогда не уточняется, как именно *позиционировать* компонент на его родительском компоненте, поскольку это легко определить по приводимому рисунку. Перечислим способы позиционирования компонента:

- перетаскивание мышью по форме (при этом на форме появляются вспомогательные *линии выравнивания*, позволяющие осуществить привязку компонента к границам формы или разместить его на уровне границ существующих компонентов);
- использование панели выравнивания **Layout** *(см. разд. 9.1)*;
- перемещение на один пиксел с помощью клавиш со стрелками;
- явное указание значения свойства `Location` в окне **Properties** (работа с окном **Properties** подробно рассматривается в *разд. 1.3*).

Размеры визуальных компонентов также обычно не уточняются. Для настройки размеров можно воспользоваться перетаскиванием мышью за один из маркеров, окружающих выделенный компонент, или клавишами со стрелками при нажатой клавише <Shift>. Можно также явно задать значения свойства `Size` в окне **Properties** или воспользоваться панелью **Layout**.

Дополнительные сведения, связанные с настройкой меню приложения, его панели инструментов и статусной панели, приводятся в *главах 18*, *21* и *22*.

1.3. Настройка свойств форм и компонентов

Для настройки свойств форм и компонентов используется *окно свойств* **Properties**, быстро перейти в которое можно с помощью комбинации клавиш <Alt>+<Enter>. Окно **Properties** обычно располагается в правом нижнем углу экрана и может находиться в двух режимах: **Properties** и **Events** (рис. 1.5). В настоящем разделе мы рассмотрим режим **Properties**, предназначенный для отображения *свойств* выделенного компонента или группы компонентов (для перехода в данный режим надо нажать третью кнопку  на панели инструментов окна свойств).

Если выделена группа компонентов, то в окне свойств отображаются только те свойства, которые имеются у всех выделенных компонентов.

Выделение группы компонентов позволяет быстро задать для них одинаковые размеры, заголовки или другие общие свойства, а также переместить их, сохраняя взаимное расположение ("как одно целое"). Перечислим способы выделения группы компонентов:

- щелчок на компонентах при нажатой клавише <Shift> или <Ctrl>;

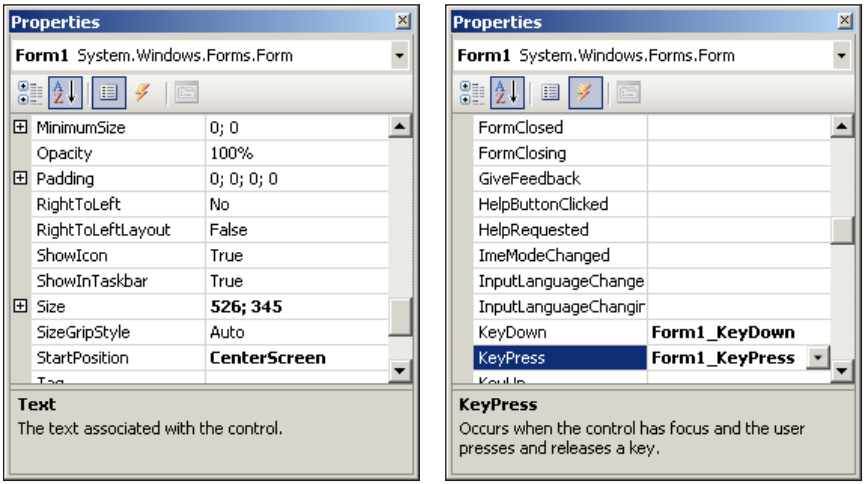


Рис. 1.5. Окно **Properties** в режимах **Properties** *(слева)* и **Events** *(справа)*

- охват компонентов пунктирной рамкой, которая появляется на форме при перемещении мыши с нажатой левой кнопкой (для выделения достаточно захватить рамкой часть компонента).

Если выделена группа компонентов, то один компонент этой группы является *текущим* (его маркеры имеют белый цвет). На рис. 1.6 приведен фрагмент формы с четырьмя выделенными компонентами; текущим является компонент `button2`. Результат некоторых действий (например, связанных с выравниванием компонентов на форме — *см. разд. 9.1*) зависит от того, какой компонент выделенной группы является текущим. Для того чтобы сделать текущим другой компонент из выделенной группы, достаточно выполнить на нем щелчок мышью. Для снятия выделения с группы компонентов надо выделить компонент, не входящий в эту группу.

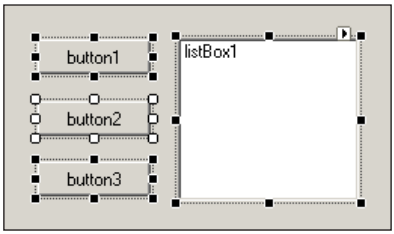


Рис. 1.6. Фрагмент формы с группой выделенных компонентов

Полезно знать, что если выделен некоторый компонент, то для выделения его родительского компонента достаточно нажать клавишу <Esc> (таким способом можно быстро выделить любой компонент-контейнер, даже если его целиком

покрывают дочерние компоненты). Для выделения формы есть еще более простой способ: щелчок мышью на ее *заголовке*.

С помощью двух первых кнопок окна свойств  (см. рис. 1.5) можно настроить способ отображения свойств: по категориям (первая кнопка) или в алфавитном порядке имен (вторая кнопка). Каждая категория имеет имя (например, **Appearance**) и содержит "родственные" свойства. Используя группировку свойств по алфавиту, проще перейти к требуемому свойству; в то же время, при начальном ознакомлении со свойствами, имеющимися у компонента, удобнее использовать группировку по категориям.

Настройку свойств форм и компонентов мы будем описывать в листингах, озаглавленных "Настройка свойств". Примером такого листинга является листинг 1.1.

В листингах настройки свойств вначале указывается имя компонента, свойства которого требуется изменить, а затем, после двоеточия, — список его настраиваемых свойств в формате

```
имя свойства = новое значение свойства
```

(значение свойства выделяется полужирным шрифтом). Свойства перечисляются через запятую.

Компоненты могут иметь *составные свойства*, т. е. свойства, имеющие собственные свойства (например, `Font`). Если требуется настроить одно или несколько свойств составного свойства, в листинге настройки свойств используется разделитель "точка" (соответствующие примеры приводятся в листинге 1.2). Составные свойства помечаются в окне **Properties** знаком "+" слева от имени (на рис. 1.5 знаком "+" помечены свойства `MinimumSize`, `Padding` и `Size`). Щелчок на знаке "+" приводит к отображению в окне всех свойств выбранного составного свойства; знак "+" при этом меняется на "–". Щелчок на знаке "–" сворачивает список свойств составного свойства.

Листинг 1.2 демонстрирует еще одно используемое обозначение: если требуется очистить какое-либо свойство, то в качестве его значения указывается курсивный текст *"пустая строка"*.

Для значений логических свойств в листингах "Настройка свойств" используются константы **`True`** и **`False`**, начинающиеся с заглавной буквы (в отличие от ключевых слов языка C# `true` и `false`); это связано с тем, что именно так указываются логические константы в окне **Properties**. Обозначения **`True`** и **`False`** будут также использоваться в тексте примеров при описании действий, связанных с окном **Properties**. Однако при комментировании фрагментов *программного кода* мы будем обозначать логические константы так, как это принято в языке C#: `true` и `false`.

Листинг 1.1. Настройка свойств (фрагмент листинга 5.1)

```
Form1: Text = Главное окно, MaximizeBox = False,
  FormBorderStyle = FixedSingle
button1: Text = Открыть подчиненное окно
button2: Text = Открыть диалоговое окно
```

Листинг 1.2. Настройка свойств (копия листинга 30.3)

```
Form2: Text = пустая строка, ControlBox = False,
  FormBorderStyle = FixedSingle, Opacity = 80%, ShowInTaskbar = False,
  StartPosition = CenterScreen, UseWaitCursor = True
label1: Text = Тригонометрические функции, AutoSize = False, Dock = Fill,
  TextAlign = MiddleCenter, Font.Name = Times New Roman, Font.Size = 32,
  Font.Bold = True
```

Для задания некоторых свойств, связанных с привязкой или выравниванием (например, `Dock` или `TextAlign`), в окне **Properties** используются *вспомогательные панели*. Для отображения таких панелей предназначена кнопка  , появляющаяся при выделении соответствующего свойства. В этих панелях надо выбрать один или несколько элементов, расположенных в требуемой позиции (слева, справа, по центру и т. п.), причем результат выбора будет отображен в окне **Properties** в виде обычного текста (например, `Bottom` или `MiddleCenter`). Хотя подобные действия являются интуитивно понятными, они, как правило, снабжаются в тексте примеров дополнительными пояснениями.

В некоторых случаях для задания свойства бывает удобно воспользоваться специальным *диалоговым окном*. Если для свойства предусмотрено такое окно, то при выделении свойства в окне **Properties** справа от него изображается кнопка с многоточием  , позволяющая вызвать диалоговое окно (примером такого свойства является `Font`).

1.4. Определение обработчиков событий

У всех компонентов имеется не только набор свойств, но и набор *событий*, с которыми можно связать методы — *обработчики событий*. Список обработчиков для выделенного компонента или группы компонентов отображается в окне свойств в режиме **Events** (см. рис. 1.5); для перехода в данный режим надо нажать четвертую кнопку  на панели инструментов окна **Properties**.

Список событий, как и список свойств, можно упорядочивать двумя способами: по категориям и по именам (в алфавитном порядке).

Двойной щелчок мышью на *пустом* поле ввода рядом с именем нужного события обеспечивает автоматическое создание заготовки для обработчика этого события. Обработчик любого события для любого компонента оформляется как метод той формы, на которой расположен данный компонент. Во всех примерах книги используются имена методов-обработчиков, предлагаемые системой Visual C# по умолчанию: это позволяет легко определить, с каким компонентом и каким событием связан обработчик.

Тексты обработчиков приводятся в книге в листингах, заголовок которых начинается со слова "Обработчик" (см. листинг 1.3, являющийся копией листинга 4.3).

Листинг 1.3. Обработчик `Form1.MouseDown`

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
  button1.Location = new Point(e.X - button1.Width / 2,
    e.Y - button1.Height / 2);
}
```

Полужирным шрифтом выделяются те строки, которые требуется добавить в автоматически созданную заготовку метода-обработчика. Иногда фрагменты программы снабжаются комментариями, хотя чаще пояснения даются в основном тексте примера или в разделе "Комментарии".

Если в качестве обработчика нужно указать имя *уже имеющегося* метода, то достаточно выбрать имя этого метода в выпадающем списке рядом с именем события в окне свойств (кнопка разворачивания списка появляется только около выделенного события; на рис. 1.5 справа эта кнопка изображена рядом с событием `KeyPress`).

Так как при разработке программ в Visual C# необходимо иметь четкое представление о том, как создаются и используются обработчики событий, этой теме посвящен специальный проект EVENTS *(см. главу 4)*. Связывание обработчика с несколькими событиями подробно обсуждается в *главе 6*.

1.5. Внесение изменений в текст программы

При выполнении примеров из данной книги изменения, как правило, вносятся в текст уже имеющихся обработчиков. Если изменения являются существенными и охватывают весь текст обработчика, то приводится его новый полный

текст, в котором измененные или добавленные строки или фрагменты строк выделяются полужирным шрифтом (см. листинг 1.4, являющийся копией листинга 6.7).

Листинг 1.4. Новый вариант метода `button1_Click`

```
private void button1_Click(object sender, EventArgs e)
{
  label1.Text = (sender as Button).Text[1].ToString();
  label2.Text = "=";
}
```

Если же изменения незначительны, а обработчик достаточно велик, то просто указывается, какие операторы надо добавить или заменить. Приведем два примера подобных исправлений.

В метод `SaveToFile` формы `Form1` добавьте оператор

```
textBox1.Modified = false;
```

В методе `button2_Click` формы `Form1` вставьте после оператора

```
Image im = new Bitmap(s);
```

два следующих оператора:

```
Graphics g = Graphics.FromImage(im);
```

```
g.Dispose();
```

Если место добавления не уточняется, то операторы должны добавляться *в конец* указанного метода.

Перейти к уже имеющемуся обработчику для его корректировки можно, либо перемещаясь по тексту файла, либо с помощью окна свойств, выполнив двойной щелчок мышью на поле ввода с именем нужного обработчика.

Аналогичным образом описываются изменения текста программы, не связанные с конкретным обработчиком. Приведем два примера.

В описание класса `Form1` добавьте поле

```
private Form2 form2 = new Form2();
```

В конструктор класса `Form1` добавьте оператор

```
AddOwnedForm(form2);
```

ПРИМЕЧАНИЕ

В качестве русского эквивалента английского термина operator в книге используется термин "операция", например, "операция +", в то время как термин "оператор" применяется для обозначения "программных инструкций" (statements).

При наборе и редактировании текста программ следует использовать дополнительные возможности редактора Visual C#. Опишем некоторые из них.

- Если ввести имя объекта (например, `button1`) и точку после него, то на экране появится список всех методов и свойств, которые имеет данный объект, причем для быстрого перехода к нужному методу достаточно набрать несколько его первых символов. Для вставки в текст программы имени выбранного метода или свойства надо нажать клавиши <Enter>, <Tab> или <Пробел>. Список методов и свойств можно вызвать и явным образом, нажав комбинацию клавиш <Ctrl>+<Пробел>.
- После ввода имени метода и скобки (на экране появляется подсказка с кратким описанием этого метода и списком всех его параметров. Если метод является *перегруженным*, т. е. может вызываться с различным набором параметров, то можно просмотреть все его перегруженные варианты, нажимая клавиши <↑> и <↓>. Для вызова подобной подсказки можно также нажать комбинацию клавиш <Ctrl>+<Shift>+<Пробел>.
- Редактор среды Visual C# позволяет автоматически *форматировать* текст программы, выполняя правильную расстановку *отступов* и *пробелов* в каждой строке кода. В частности, форматирование операторов, заключенных в блок {}, выполняется при вводе фигурной скобки }, завершающей этот блок. Кроме того, предусмотрена команда, позволяющая отформатировать весь текст, содержащийся в файле; для этого достаточно последовательно нажать две комбинации клавиш: <Ctrl>+<K> и <Ctrl>+<D>. Следует заметить, что форматирование выполняется только для синтаксически правильного программного кода, т. е. кода, в котором не выделен как ошибочный ни один из его фрагментов (ошибочные фрагменты кода подчеркиваются красной волнистой линией). Во всех листингах, приводимых в книге, сохранено форматирование, выполненное редактором среды Visual C#; изменен лишь *размер отступа*: вместо 4 пробелов, установленных в редакторе по умолчанию, используется отступ, равный 2 пробелам (это позволяет разместить в одной строке листинга операторы, которые при стандартной величине отступа потребовалось бы разбивать на две строки).
- Для быстрого перехода к нужному фрагменту программы удобно использовать *закладки* (bookmarks). Для установки/отмены закладки на текущей строке программы (т. е. строке, содержащей клавиатурный курсор) надо нажать комбинацию клавиш <Ctrl>+<B> и затем клавишу <T>, а для перехода

к следующей или предыдущей установленной закладке надо нажать комбинацию клавиш <Ctrl>+<B> и затем клавишу <N> или <P> соответственно. Можно также использовать меню **Edit | Bookmarks** и кнопки быстрого доступа  на панели **Text Editor**.

- Если требуется *закомментировать* какой-либо фрагмент программы, то достаточно выделить его и нажать комбинацию клавиш <Ctrl>+<E>, а затем клавишу <C>. Для того чтобы раскомментировать закомментированный фрагмент, надо выделить его и нажать комбинацию клавиш <Ctrl>+<E>, а затем клавишу <U>. Если требуется закомментировать или раскомментировать одну строку, то вместо ее выделения достаточно установить на ней клавиатурный курсор, после чего нажать указанные клавиши. Вместо клавиатурных комбинаций можно использовать кнопки быстрого доступа  на панели **Text Editor**.
- Редактор среды Visual C# обладает богатыми возможностями *поиска и замены*. Эти возможности подробно описываются в комментарии 2 к *разд. 10.7*.
- Наконец, следует упомянуть о возможности *автогенерации кода* с помощью специальных шаблонов (code snippets). Использование подобных шаблонов описывается в *разд. 31.1* и комментарии к нему.

1.6. Запуск приложения

Каждый этап разработки проекта описывается в отдельном разделе главы, посвященной этому проекту. В любом разделе вначале перечисляются необходимые действия, связанные с модификацией проекта. Затем следует абзац, начинающийся со слова **Результат**. В этом абзаце описывается, как будет работать новый вариант программы. Появление абзаца **Результат** служит признаком того, что модифицированный проект можно откомпилировать и запустить на выполнение (для этого достаточно нажать клавишу <F5> или кнопку ).

Если код программы был набран с синтаксическими ошибками, то сообщения об этих ошибках появляются в окне **Error List**, которое при этом становится активным. Для того чтобы перейти на строку программы, в которой обнаружена первая синтаксическая ошибка, достаточно нажать клавишу <↓> (в результате сообщение о первой ошибке будет подсвечено), после чего нажать клавишу <Enter>. Можно также выполнить двойной щелчок мышью на строке с сообщением об ошибке.

При компиляции программы в среде Visual C# важную роль играют две настройки среды, расположенные в группе **Projects and Solutions** окна **Options** (данное окно вызывается командой меню **Tools | Options...**). Первая настрой-

ка — флажок **Always show Error List if build finishes with errors** (Всегда отображать окно Error List, если компиляция завершилась с ошибками) в разделе **General**. Этот флажок должен быть установлен. Вторая настройка — выпадающий список **On Run, when build or deployment errors occur** (Если обнаружены ошибки компиляции или размещения при выполнении команды Run) в разделе **Build and Run**. В этом списке должен быть выбран вариант **Do not launch** (Не запускать программу). В экспресс-варианте среды Visual C# для отображения указанных настроек необходимо установить флажок **Show all settings** в окне **Options** (по умолчанию этот флажок не установлен).

Если после абзаца **Результат** следует абзац с пометкой **Ошибка**, значит, программа, несмотря на успешную компиляцию, будет работать не совсем правильно, и в нее надо внести дополнительные исправления (таким способом в примерах привлекается внимание к типичным ошибкам, которые могут возникнуть в аналогичных ситуациях). Если после абзаца **Результат** следует абзац с пометкой **Недочет**, значит, программа работает правильно и делает то, что требуется, но имеет дефекты интерфейса, т. е. ею неудобно пользоваться. Как правило, сразу после описания ошибки или недочета указывается способ их исправления, хотя иногда исправление откладывается до следующего раздела.

Глава 2

Консольное приложение: проект DISKINFO

Проект DISKINFO знакомит с приемами разработки консольных приложений. Описывается структура консольного приложения и подключаемые к нему пространства имен. Рассматривается форматированный вывод, в частности, использование управляющих последовательностей. Описываются классы `DriveInfo`, `StringBuilder` и `Environment`. Кроме того, описываются приемы работы с параметрами командной строки.

2.1. Создание консольного приложения

При создании нового проекта — консольного приложения следует выполнять практически те же действия, что и при создании Windows-приложения *(см. разд. 1.1)*. Единственным отличием является указание другого типа проекта в окне **New Project**: для консольного приложения следует выбрать вариант **Console Application**.

Созданный проект будет содержать файл Program.cs, который сразу загрузится в редактор (листинг 2.1).

Листинг 2.1. Исходный текст файла Program.cs для консольного приложения

```
using System;
using System.Collections.Generic;
using System.Text;

namespace DISKINFO
{
  class Program
  {
```

```
    static void Main(string[] args)
    {
    }
  }
}
```

Первые три оператора в файле Program.cs содержат названия трех пространств имен, классы из которых используются наиболее часто (см. комментарий 1). Благодаря этим операторам, имена классов можно указывать в программе без уточнения, к какому пространству имен они принадлежат (например, вместо `System.Console` можно писать просто `Console`).

Метод `Main` является стартовой точкой выполнения программы. Его параметр `args` позволяет получить информацию о *параметрах командной строки*, указанных при запуске данной программы (использование параметра `args` демонстрируется в *разд. 2.3*).

Добавьте в метод `Main` новые операторы (листинг 2.2).

Листинг 2.2. Добавление к методу `Main`

```
Console.WriteLine("Программа DISKINFO\n");
Console.WriteLine("\nДля завершения программы нажмите <Enter>...");
Console.ReadLine();
```

Результат. При запуске программы на экране появляется особое *консольное окно*, используемое для ввода/вывода данных в текстовом режиме. Это окно обладает всеми свойствами окна приложения DOS; в частности, оно имеет расширенное системное меню, вызываемое комбинацией клавиш <Alt>+<Пробел>. При выполнении программы можно переключаться в полноэкранный режим комбинацией клавиш <Alt>+<Enter> (эта же комбинация восстанавливает оконный режим). Консольное окно будет содержать текст, приведенный в листинге 2.3.

После завершения программы консольное окно немедленно закрывается. Чтобы можно было ознакомиться с содержимым окна, в программу был добавлен вызов метода `ReadLine` класса `Concole` (см. листинг 2.2, последний оператор). Этот метод предназначен для ввода строки, причем признаком завершения ввода является нажатие клавиши <Enter>. Поэтому до нажатия клавиши <Enter> консольное окно будет оставаться на экране. Строка, возвращаемая методом `ReadLine`, в программе не используется, поэтому сохранять ее в какой-либо переменной нет необходимости.

Листинг 2.3. Содержимое консольного окна при выполнении программы

```
Программа DISKINFO


Для завершения программы нажмите <Enter>...
```

ПРИМЕЧАНИЕ

Следует отметить, что в консольном окне русский текст будет отображаться правильно, несмотря на то, что в этом окне используется особая DOS-кодировка OEM 866 (MS-DOS Russian), в которой коды русских букв отличаются от кодов в стандартной Windows-кодировке ANSI 1251 (Windows Cyrillic). Перекодирование текста при выполнении консольных приложений в .NET выполняется автоматически, при этом учитываются языковые настройки Windows (в нашем случае — настройки для русского языка).

Комментарии

1. Дадим краткую характеристику пространствам имен, автоматически подключенным к нашему приложению (см. листинг 2.1, первые три оператора). Пространство имен `System` является основным пространством стандартной библиотеки .NET Framework; оно содержит определения классов, необходимых практически в любой программе. В частности, в нем определен класс `Console`, обеспечивающий ввод/вывод для консольного окна. Пространство имен `System.Collections.Generic` содержит *обобщенные* классы-коллекции. Классы-обобщения (generics) появились в версии .NET 2.0. Способ реализации коллекций, основанный на обобщениях, позволяет при описании коллекции указать, данные какого типа будут в ней содержаться (пример использования обобщенного класса-коллекции приводится в *разд. 7.1*). Пространство имен `System.Text` содержит различные классы и перечисления, предназначенные для обработки строковых данных; в *разд. 2.2* будет использован класс `StringBuilder`, определенный в этом пространстве имен.
2. Комбинация `\n`, использованная в строковых константах (см. листинг 2.2), является одной из *управляющих последовательностей (escape-последовательностей)*, которые можно указывать в строковых выражениях. Эта комбинация обозначает символ с кодом 10 (переход на новую строку). Среди других управляющих последовательностей отметим `\\` (символ \), `\"` (двойная кавычка), `\'` (одинарная кавычка), `\0` (символ с кодом 0), `\r` (возврат каретки — символ с кодом 13), `\b` (символ с кодом 8, генерируемый клавишей <Backspace>), `\t` (табуляция — символ с кодом 9) и `\u`*N* (символ

Unicode с шестнадцатеричным кодом *N*). В управляющей последовательности \u*N* код *N* должен состоять из 4 цифр и может иметь значение от 0000 до FFFF; например, символ \u0041 обозначает латинскую букву A (десятичный код 65).

3. Если запустить консольное приложение не в режиме **Debug** (клавишей <F5>), а в режиме **Release** (комбинацией клавиш <Ctrl>+<F5>), то при его завершении в консольном окне появится текст "Для продолжения нажмите любую клавишу..." и окно будет закрыто только после нажатия произвольной клавиши. Таким образом, в этом режиме нет необходимости в операторе Console.ReadLine (более того, при наличии этого оператора для закрытия приложения в режиме **Release** придется вначале нажать клавишу <Enter>, а затем еще одну произвольную клавишу). Однако при запуске полученного exe-файла непосредственно из Windows (а не из среды Visual C#) приложение выполняется так же, как в режиме **Debug**, т. е. сообщение "Для продолжения нажмите любую клавишу..." не возникает.

2.2. Получение информации о текущем диске

В начало файла Program.cs добавьте оператор

```
using System.IO;
```

В класс Program добавьте новый метод DInfo (листинг 2.4) и измените метод Main (листинг 2.5).

Листинг 2.4. Метод DInfo класса Program

```
static void DInfo(string path)
{
  string none = "---",
    d = path[0].ToString().ToUpper();
  DriveInfo di = new DriveInfo(d);
  StringBuilder s = new StringBuilder(40);
  s.AppendFormat(" {0,-4}", d);
  if (di.DriveType != DriveType.NoRootDirectory)
  {
    s.AppendFormat(" {0,-9}", di.DriveType);
    if (di.IsReady)
      s.AppendFormat("{0,12:N0} {1,12:N0}", di.TotalSize / 1024,
```

```
        di.TotalFreeSpace / 1024);
    else
      s.AppendFormat("{0,12} {0,12}", none);
  }
  else
    s.AppendFormat(" {0,-9}{0,12} {0,12}", none);
  Console.WriteLine(s);
}
```

Листинг 2.5. Новый вариант метода `Main`

```
static void Main(string[] args)
{
  Console.WriteLine("Программа DISKINFO\n");
  Console.WriteLine(" Disk Type         Size (K)     Free (K)");
  Console.WriteLine(new String('=', 40));
  DInfo(Environment.CurrentDirectory);
  Console.WriteLine("\nДля завершения программы нажмите <Enter>...");
  Console.ReadLine();
}
```

Результат. При запуске программы в консольном окне отображается информация о текущем диске (листинг 2.6).

Листинг 2.6. Содержимое консольного окна при выполнении программы

```
Программа DISKINFO

 Disk Type         Size (K)     Free (K)
========================================
 D    Fixed      16 370 256    4 292 728

Для завершения программы нажмите <Enter>...
```

Комментарии

1. Метод `DInfo` в качестве параметра принимает строку `path`, из которой используется только первый символ (предполагается, что данный символ является *буквой диска*). Поскольку выражение `path[0]` имеет тип `char`, для

возможности использования этого выражения в качестве строкового параметра конструктора класса `DriveInfo` необходимо выполнить его явное преобразование к типу `string` с помощью метода `ToString`. Для повышения наглядности полученная односимвольная строка затем преобразуется к верхнему регистру с помощью метода `ToUpper` класса `string`.

2. Для получения информации о диске используется класс `DriveInfo`, появившийся в версии .NET 2.0 и определенный в пространстве имен `System.IO`. Для создания объекта класса `DriveInfo` достаточно вызвать его конструктор, указав в нем букву требуемого диска. Свойство `DriveType` (типа перечисления `DriveType`) позволяет определить тип дискового носителя, причем даже в случае, если диск недоступен (среди возможных значений перечисления `DriveType` укажем `Removable`, `Fixed`, `Network` и `CDRom`). Для доступного диска можно определить его размер в байтах (свойство `TotalSize` типа `long`), размер свободного пространства в байтах (свойство `TotalFreeSize` типа `long`) и его метку (строковое свойство `VolumeLabel`, которое доступно и для чтения, и для записи).

   Заметим, что тип `long`, используемый для свойств `TotalSize` и `TotalFreeSize`, позволяет хранить числа в диапазоне от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 (таким образом, имеется принципиальная возможность определять размер диска, содержащего более 9 млн терабайтов).

3. Для формирования строки с информацией о диске используется объект `s` класса `StringBuilder`. Этот класс определен в пространстве имен `System.Text` и позволяет более эффективно (по сравнению с классом `string`) выполнять операции, связанные с преобразованием строк. В конструкторе класса `StringBuilder` мы указали *емкость* формируемой строки (т. е. ее максимально возможный размер, равный 40 символам), однако это не означает, что класс `s` не может содержать строки большего размера: если размер фактической строки, помещаемой в объект типа `StringBuilder`, превышает его емкость, то происходит автоматическое удвоение емкости. Явное указание емкости в конструкторе лишь позволяет избежать лишних операций по выделению и освобождению памяти.

   Важной особенностью объектов типа `StringBuilder`, по сравнению с объектами типа `string`, является доступность их символов не только на чтение, но и на запись. Кроме того, любые действия по изменению строки типа `StringBuilder` выполняются над этой же строкой и не приводят к появлению новой (измененной) строки, как это происходит в методах класса `string`, связанных с изменением строковых данных. Использованный в программе метод `AppendFormat` добавляет к прежнему содержимому строки новые данные, предварительно выполняя их форматирование.

4. При форматировании данных в методе `AppendFormat` используются специальные *форматные настройки*, имеющие в общем случае вид `{ind, width :spec}`, где `ind` определяет индекс форматируемого элемента в последующем списке параметров (индексирование проводится от 0), а `width` определяет минимальную ширину поля вывода для форматируемого элемента и способ его выравнивания в пределах данного поля: если значение `width` положительно, то выполняется выравнивание по правой границе поля вывода, а если отрицательно — то по левой. Атрибут `spec` задает *спецификатор формата* для данного элемента. Использованный в программе спецификатор `N` позволяет выводить число с пробелами — разделителями тысяч, а указанный после него `0` означает нулевое количество дробных знаков (по умолчанию в формате `N` выводятся два дробных знака). Из других спецификаторов формата отметим `C` (денежный формат), `D` (десятичный целочисленный формат), `X` (шестнадцатеричный целочисленный формат), `E` (экспоненциальный числовой формат), `F` (числовой формат с фиксированной точкой), `P` (процентный формат). Спецификаторы формата можно использовать во многих методах, связанных с формированием и выводом строк, например, в методе `Format` класса `string` и методах `Write` и `WriteLine` класса `Console`. Из атрибутов, входящих в форматные настройки, только атрибут `ind` является обязательным; при этом допустимо указывать несколько *одинаковых* значений атрибута `ind`, если требуется вывести один и тот же элемент данных несколько раз (см. два последних вызова метода `AppendFormat` в листинге 2.4). Если отсутствует атрибут `width`, то используется ширина поля вывода, минимально необходимая для отображения отформатированного элемента. Если отсутствует атрибут `spec`, то выбирается вариант форматирования по умолчанию (этот вариант соответствует спецификатору формата `G`). При отсутствии `width` предшествующая запятая в форматных настройках не указывается; при отсутствии `spec` не указывается предшествующее двоеточие.

5. Вместо явного задания строки, содержащей 40 символов = (знак равенства), мы воспользовались вариантом конструктора `string(c, n)`, позволяющим создать строку указанной длины `n`, состоящую из одинаковых символов `c`.

6. Для определения текущего диска использовалось свойство `CurrentDirectory` класса `Environment`, позволяющее получить текущий каталог для данного приложения и доступное как для чтения, так и для записи. Среди других полезных свойств этого класса отметим `CommandLine`, возвращающее строку с полным именем запущенного exe-файла и следующими за ним параметрами командной строки, `SystemDirectory`, возвращающее полное имя системного каталога Windows, а также набор свойств, позволяющих получить информацию о компьютере, пользователе и операционной системе: `MachineName`,

`UserName`, `OSVersion`, `Version` (последнее свойство возвращает полный номер используемой версии .NET Framework). Есть также свойство-массив `GetCommandLineArgs` типа `string[]`, первый элемент которого (с индексом `0`) содержит полное имя exe-файла, а каждый последующий элемент — очередной параметр командной строки (таким образом, свойство-массив `GetCommandLineArgs` представляет собой строку `CommandLine`, разобранную на составные элементы).

Кроме того, класс `Environment` позволяет получить и изменить значения различных *переменных окружения*, определенных в операционной системе. Для этого он имеет методы `GetEnvironmentVariables` (возвращает массив всех переменных окружения в виде пары строк *имя–значение*), `GetEnvironmentVariable(name)` (возвращает строковое значение для переменной окружения с именем `name` или `null`, если указанная переменная окружения не существует) и `SetEnvironmentVariable(name, val)` (задает переменной окружения с именем `name` новое строковое значение `val`). Если указанная в методе `SetEnvironmentVariable` переменная `name` ранее не существовала, то она создается; если параметр `val` является пустой строкой или равен `null`, то существующая переменная `name` уничтожается.

2.3. Использование параметров командной строки

Измените метод `Main` в файле Program.cs (листинг 2.7).

Листинг 2.7. Новый вариант метода `Main`

```
static void Main(string[] args)
{
  Console.WriteLine("Программа DISKINFO\n");
  Console.WriteLine(" Disk Type         Size (K)     Free (K)");
  Console.WriteLine(new String('=', 40));
  if (args.Length == 0)
    DInfo(Environment.CurrentDirectory[0]);
  else
    foreach (string d in args)
      DInfo(d);
  Console.WriteLine("\nДля завершения программы нажмите <Enter>...");
  Console.ReadLine();
}
```

Результат. Если в качестве параметров командной строки указано одно или несколько имен дисков, то выводится информация об этих дисках (параметры командной строки должны отделяться друг от друга пробелами). Если параметры не указаны, то выводится информация о текущем диске. Для задания параметров командной строки в среде Visual C# следует выполнить команду **Project | DISKINFO Properties...**, в загруженном в редактор списке свойств проекта выбрать раздел **Debug** и ввести параметры командной строки в поле **Command line arguments**. Так, если в строке параметров указать **a d f z**, то в результате выполнения программы будет выведен текст, подобный приведенному в листинге 2.8.

В системе Windows программу с параметрами можно запустить, например, из меню **Пуск** командой **Выполнить...**:

```
C:\CSProjects\DISKINFO\bin\Debug\DISKINFO.exe  a  d  f  z
```

Если программа запускается с помощью *ярлыка* (shortcut), то ее параметры командной строки можно задать в окне свойств, которое вызывается командой **Свойства** контекстного меню ярлыка. В окне свойств надо перейти на вкладку **Ярлык** и указать нужные параметры в поле **Файл** (или **Объект**).

Листинг 2.8. Содержимое консольного окна при выполнении программы

```
Программа DISKINFO

 Disk Type         Size (K)     Free (K)
=======================================
 A    Removable         ---          ---
 D    Fixed      16 370 256    4 285 224
 F    Fixed       6 309 192      543 420
 Z    ---               ---          ---

Для завершения программы нажмите <Enter>...
```

Ошибка. Если в качестве одного из параметров командной строки указать строку, начинающуюся не с латинской буквы, то при попытке обработать этот параметр возникнет *ошибка времени выполнения*, которая в случае запуска программы непосредственно в системе Windows приведет к аварийному завершению программы. Если программа была запущена из среды Visual C#, то произойдет возврат в среду Visual C# и в тексте программы будет выделен оператор, выполнение которого привело к возникновению ошибки. В последнем случае для завершения программы достаточно нажать комбинацию клавиш

<Shift>+<F5> или кнопку с изображением синего квадратика  (более подробно действия при возникновении ошибок времени выполнения описываются в *разд. 3.1*).

Исправление. В методе `DInfo` перед оператором

```
DriveInfo di = new DriveInfo(d);
```

добавьте следующий фрагмент:

```
if (d[0] < 'A' || d[0] > 'Z')
  return;
```

Результат. Теперь программа обрабатывает только те параметры, которые начинаются с латинской буквы (прописной или строчной).

> ***ПРИМЕЧАНИЕ***
>
> Другим способом исправления отмеченной ошибки является явная обработка возникающей исключительной ситуации *(исключения)* в блоке `try-catch` *(см. главу 3)*. Однако если имеется возможность исправить ошибку без привлечения механизма обработки исключений, то этой возможностью следует воспользоваться, поскольку обработка исключений существенно замедляет выполнение программы.

Комментарий

Для определения количества параметров командной строки достаточно воспользоваться свойством `Length` массива `args` (см. листинг 2.7). Для перебора всех параметров мы воспользовались циклом `foreach`, на каждой итерации которого переменная `d` получает значение очередного элемента массива `args`. Заметим, что изменять элементы массива с помощью цикла `foreach` нельзя.

Глава 3

Обработка исключений: проект EXCEP

Проект EXCEP знакомит с приемами обработки исключительных ситуаций (исключений). Описывается структура try-блоков и демонстрируются особенности, связанные с использованием вложенных try-блоков. Дается обзор исключений, связанных с арифметическими операциями, рассматриваются операции checked и unchecked. Описываются действия, которые можно выполнить при возникновении исключения в различных режимах запуска приложения. Рассматривается метод Parse преобразования строки в число и оператор throw возбуждения исключения (в том числе повторного). Описывается вариант try-блока, использующий раздел finally.

3.1. Обработка конкретного исключения и групп исключений

В этом примере, как и в предыдущем, будет разрабатываться консольное приложение. Создайте заготовку проекта для консольного приложения *(см. разд. 2.1)* и измените описание класса Program в файле Program.cs (листинг 3.1).

Листинг 3.1. Описание класса Program

```
class Program
{
  static void M1(int x, int y, int z)
  {
    try
    {
      int a = checked((int)Math.Pow(x, y));
      Console.WriteLine("x ^ y / z = {0}", a / z);
    }
    catch (DivideByZeroException)
```

```
    {
      Console.WriteLine("DivideByZero Exception");
    }
    Console.WriteLine("M1 finished");
  }
  static void M2(int x, int y, int z)
  {
    try
    {
      M1(x, y, z);
    }
    catch (ArithmeticException)
    {
      Console.WriteLine("Arithmetic Exception");
    }
    Console.WriteLine("M2 finished");
  }
  static void Main(string[] args)
  {
    Console.Write("x = ");
    int x = int.Parse(Console.ReadLine());
    Console.Write("y = ");
    int y = int.Parse(Console.ReadLine());
    Console.Write("z = ");
    int z = int.Parse(Console.ReadLine());
    M2(x, y, z);
    Console.ReadLine();
  }
}
```

Результат. Для трех введенных целых чисел *x*, *y*, *z* программа вычисляет выражение x^y/z, обрабатывая возникающие при этом *исключительные ситуации* (исключения, exceptions). Для выхода из программы надо нажать клавишу <Enter>. Опишем различные варианты работы программы.

Вариант A. Обработка допустимых значений (листинг 3.2). Вычисления выполняются успешно; ни один обработчик исключений не активизируется.

Листинг 3.2. Содержимое консольного окна при обработке допустимых значений

```
x = 9
y = 2
z = 3
x ^ y / z = 27
M1 finished
M2 finished
```

Вариант B. Деление на 0 (листинг 3.3). Активизируется обработчик try-блока метода M1, обрабатывающий исключение типа DivideByZeroException, после чего выполнение программы продолжается с оператора, следующего за данным try-блоком.

Листинг 3.3. Содержимое консольного окна при делении на 0

```
x = 1
y = 1
z = 0
DivideByZero Exception
M1 finished
M2 finished
```

Вариант C. Целочисленное переполнение (листинг 3.4). При попытке возведения числа 10 в степень 10 (и последующего преобразования результата к целому типу) возникает исключение OverflowException. Поскольку в разделе catch try-блока метода M1 обработка исключения OverflowException не предусмотрена, происходит немедленный переход в обработчик try-блока следующего уровня (т. е. в раздел catch try-блока метода M2). Здесь исключение OverflowException обрабатывается, так как оно является *потомком* исключения ArithmeticException — предка всех исключений, порожденных ошибками при выполнении арифметических операций. После этой обработки выполнение программы продолжается с оператора, следующего за try-блоком метода M2.

Листинг 3.4. Содержимое консольного окна при целочисленном переполнении

```
x = 10
y = 10
z = 1
Arithmetic Exception
M2 finished
```

Вариант D. Ввод недопустимого символа. После ввода недопустимого символа (например, звездочки *) выполнение программы *немедленно прерывается,* происходит возврат в среду Visual C# и в тексте программы выделяется оператор, выполнение которого привело к возникновению исключения (в нашем случае это будет первый из операторов метода `Main`, содержащих вызов функции `Parse`). Такое поведение программы объясняется тем, что в методе `Main` не предусмотрена обработка возникшего исключения `FormatException`, и поэтому активизируется режим обработки исключения по умолчанию.

В данной ситуации возможны два варианта действий.

1. Немедленно прервать выполнение программы, нажав комбинацию клавиш <Shift>+<F5> или кнопку с изображением синего квадратика  .
2. Продолжить выполнение программы, пропустив ошибочный оператор и, возможно, несколько следующих операторов. Для этого надо зацепить мышью желтую стрелку  , расположенную рядом с ошибочным оператором, и перетащить ее вниз к тому оператору, с которого надо продолжить выполнение программы, после чего нажать клавишу <F5> или кнопку с изображением зеленого треугольника  . Можно также перейти к *пошаговому выполнению программы*, нажимая кнопку **Step Into**  (или клавишу <F11>) или кнопку **Step Over**  (или клавишу <F10>). Любая из этих кнопок обеспечивает выполнение текущего оператора (т. е. оператора, на который указывает желтая стрелка). Отличие между ними состоит в том, что если текущий оператор является вызовом функции и код этой функции доступен, то кнопка **Step Into** обеспечивает переход на начало этой функции, а кнопка **Step Over** немедленно выполняет функцию и переходит к следующему за ней оператору.

ПРИМЕЧАНИЕ

Поведение программы, описанное в ситуации D, соответствует обработке исключений по умолчанию в режиме **Debug** (т. е. при запуске программы с помощью клавиши <F5>). Если программа запущена в режиме **Release** (т. е. с помощью комбинации клавиш <Ctrl>+<F5>), то при возникновении исключения на экране

появляется диалоговое окно с информацией о возникшем исключении, содержащее две кнопки: **Continue** (для продолжения выполнения программы) и **Quit** (для ее немедленного завершения). В подобной ситуации не происходит перехода в редактор среды Visual C# и выделения ошибочного оператора.

Комментарии

1. Использованный в программе метод `Parse` для типа `int` (см. листинг 3.1) позволяет преобразовать указанную строку к типу `int` (для успешности подобного преобразования необходимо, чтобы в строке содержалось изображение некоторого целого числа, возможно, дополненное слева и справа пробелами). Аналогичный метод имеется и у других числовых типов, в частности, у вещественного типа `double`. Следует иметь в виду, что при преобразовании строки в вещественное число символ разделителя дробной части определяется настройками операционной системы Windows, поэтому в программе, выполняемой в русской версии Windows, в качестве десятичного разделителя необходимо вводить *запятую*. Дополнительные сведения о языковых настройках и их изменении приводятся в комментарии 3 в *разд. 6.4*.

   Если параметр метода `Parse` невозможно преобразовать к указанному числовому типу, то возбуждается исключение типа `FormatException` (см. выше описание варианта D).

2. Добиться исключения `OverflowException` при выполнении операций с *вещественными* числами невозможно: если получающееся число окажется слишком большим, то будет возвращено особое значение типа `double` — `double.PositiveInfinity` (положительная бесконечность), с которым можно выполнять различные действия так же, как и с обычными числами. Имеются еще два особых вещественных значения: `double.NegativeInfinity` — отрицательная бесконечность и `double.NaN` — не число. Стандартные математические функции, определенные в классе `Math`, могут возвращать как "обычные", так и особые числовые значения. Например, `Math.Sqrt(-1)` (квадратный корень) вернет `double.NaN`, а `Math.Log(0)` (натуральный логарифм) вернет `double.NegativeInfinity`. Заметим, что в нашей программе мы воспользовались функцией `Pow` из класса `Math`, позволяющей выполнить возведение в степень, причем переполнение, описанное в варианте C, возникало не при вычислении этой функции, а при попытке преобразования полученного результата к типу `int` (т. е. имело место *целочисленное переполнение*).

3. Целочисленное переполнение тоже не всегда приводит к возбуждению исключения. При стандартных настройках проекта в подобной ситуации

исключение не возбуждается, а слишком большое целочисленное значение урезается посредством отбрасывания лишних старших байтов. Если такое поведение при целочисленном переполнении нежелательно (как в нашем случае), то можно включить явный контроль за переполнением, заключив опасное выражение в конструкцию `checked` с круглыми скобками (см. листинг 3.1, метод `M1`). Конструкцию `checked` можно использовать в виде *операции* для защиты выражений (как в методе `M1`), а также в виде *оператора* для защиты группы операторов; в этом случае ее синтаксис следующий:

```
checked { операторы }
```

Следует, однако, иметь в виду, что если среди указанных операторов имеются вызовы методов, то внутри этих методов контроль за целочисленным переполнением проводиться не будет.

Предусмотрена также парная конструкция `unchecked`, отключающая контроль за целочисленным переполнением. Благодаря отключенному контролю, операции над целочисленными данными выполняются существенно быстрее.

Установить контроль целочисленного переполнения на уровне всего проекта можно, изменив настройки проекта. Для этого надо выполнить команду меню **Project** | ***<Имя проекта>*** **Properties...**, перейти в появившейся вкладке с именем проекта в раздел **Build**, нажать кнопку **Advanced** и установить в появившемся окне флажок **Check for arithmetic overflow/underflow**.

3.2. Обработка любого исключения

Измените метод `Main` в файле Program.cs (листинг 3.5). Теперь программа содержит три вложенных `try`-блока.

Листинг 3.5. Новый вариант метода `Main`

```
static void Main(string[] args)
{
  try
  {
    Console.Write("x = ");
    int x = int.Parse(Console.ReadLine());
    Console.Write("y = ");
    int y = int.Parse(Console.ReadLine());
    Console.Write("z = ");
```

```
    int z = int.Parse(Console.ReadLine());
    M2(x, y, z);
  }
  catch
  {
    Console.WriteLine("Other exception");
  }
  Console.ReadLine();
}
```

Результат. В любом из вариантов A, B, C, рассмотренных в *разд. 3.1*, результат работы программы будет прежним. В варианте D (ввод недопустимого символа) в консольном окне будет выведено:

```
x = *
Other exception
```

и программа будет ожидать нажатия клавиши <Enter> для своего завершения. Таким образом, никакое исключение теперь не приведет к аварийному прерыванию выполнения программы.

Недочет. Информация, выводимая на экран, не позволяет определить, какое именно исключение возникло в ходе выполнения программы.

Исправление. Измените раздел `catch` в методе `Main`:

```
catch(Exception ex)
{
  Console.WriteLine(ex.GetType().Name + ":\n  " + ex.Message);
}
```

Результат. Теперь в случае ввода недопустимого символа будет выведена более содержательная информация:

```
x = *
FormatException:
  Input string was not in a correct format.
```

Комментарий

В последнем варианте программы был определен обработчик для исключения `Exception` — общего предка всех исключений. Поэтому он активизируется при возникновении любого исключения, не обработанного в предыдущих `try`-блоках. Использование переменной `ex` типа `Exception` позволяет получить

доступ к методам и свойствам возникшего исключения. Имя класса-исключения можно получить с помощью выражения `ex.GetType().Name`, краткое описание ошибки содержится в свойстве `Message` объекта `ex`.

3.3. Повторное возбуждение обработанного исключения

Дополните раздел `catch` для `try`-блока метода `M2`:

```
catch (ArithmeticException)
{
  Console.WriteLine("Arithmetic Exception");
  throw;
}
```

Результат. В любом из вариантов A, B, D результат работы программы будет прежним. В варианте C (целочисленное переполнение) в окне будет выведена более содержательная информация (листинг 3.6). Это связано с тем, что добавленный в раздел `catch` метода `M2` оператор `throw` выполняет *повторное возбуждение* только что обработанного исключения. Повторно возбужденное исключение окончательно обрабатывается в разделе `catch` метода `Main` *(см. разд. 3.2)*.

Листинг 3.6. Содержимое консольного окна при целочисленном переполнении

```
x = 10
y = 10
z = 1
Arithmetic Exception
OverflowException:
  Arithmetic operation resulted in an overflow.
```

Комментарии

1. Оператор `throw` используется также для явного возбуждения исключения в программе. Например, если в некотором методе `M` параметр `n` целого типа может принимать только значения от `1` до `nMax`, то при нарушении этого условия следует возбудить в методе `M` исключение `ArgumentOutOfRangeException`:

   ```
   if (n < 1 || n > nMax)
     throw new ArgumentOutOfRangeException("n");
   ```

Использованный вариант конструктора класса `ArgumentOutOfRangeException` позволяет указать имя ошибочного параметра в свойстве `Message` созданного исключения. В нашем случае данное свойство будет содержать текст: `Specified argument was out of the range of valid values. Parameter name: n` (Указанный аргумент находится вне диапазона допустимых значений. Имя параметра: n). Пример использования оператора `throw` приводится также в *разд. 30.1*.

2. В `try`-блоке может содержаться несколько разделов `catch`, каждый из которых будет обрабатывать исключения определенного типа. Предусмотрен также дополнительный раздел `try`-блока с именем `finally`, который располагается после всех разделов `catch` и содержит код, выполняющий очистку ресурсов, выделенных в программе, и другие завершающие действия. Раздел `finally` выполняется *всегда*: и при нормальной работе операторов в блоке `try`, и при возбуждении исключения, причем даже в ситуации, если это исключение не было перехвачено в предыдущих разделах `catch`. Варианты `try`-блока с разделом `finally` приводятся в *разд. 13.6—13.7*.

Глава 4

События: проект EVENTS

Проект EVENTS знакомит с базовыми приемами разработки программ, управляемых событиями (events). Демонстрируется связывание события с обработчиком (в режиме дизайна и на программном уровне), отключение обработчика от события и его повторное подключение. Рассматривается класс `Random` и свойства визуальных компонентов, связанные с их размером и положением на экране. Дается обзор структуры графического Windows-приложения и входящих в него элементов.

4.1. Связывание события с обработчиком

Проект EVENTS является первым графическим приложением, рассматриваемым в книге, поэтому действия по его разработке мы опишем более подробно *(см. также главу 1)*.

После создания нового проекта типа **Windows Application** разместите в форме `Form1` компонент-кнопку `Button`, используя *окно компонентов* **Toolbox** (проще всего выбрать компонент `Button` из группы **All Windows Forms**, содержащей все компоненты в алфавитном порядке). Добавленной кнопке будет автоматически присвоено имя `button1`.

Настройте свойства формы `Form1` и кнопки `button1` (листинг 4.1). Для этого надо использовать *окно свойств* **Properties**.

По рис. 4.1 настройте размеры формы и расположение кнопки.

С событием `Click` компонента `button1` свяжите обработчик (листинг 4.2). Для этого надо выделить на форме кнопку `button1`, например, щелкнув на кнопке мышью (в результате вокруг кнопки будут изображены маркеры, как на рис. 4.1, а окно свойств **Properties** будет настроено на отображение свойств кнопки). Затем надо выбрать в окне **Properties** режим **Events** (нажав кнопку с изображением молнии ) и выполнить двойной щелчок мышью на пустом поле ввода справа от метки **Click**. В результате в редактор среды Visual C#

будет загружен файл Form1.cs с описанием класса `Form1`, и в этот файл будет добавлена заготовка для обработчика события `Click` (метод `button1_Click`). Теперь, используя редактор, надо ввести в эту заготовку нужные операторы (в данном случае — вызов метода `Close`). Текст метода `button1_Click`, создаваемый автоматически, изображается в листинге обычным шрифтом, а текст, который надо добавить в метод, — полужирным.

Аналогичными действиями создайте обработчик события `MouseDown` для формы `Form1` (листинг 4.3). Поскольку это событие должно быть связано не с кнопкой, а с формой, предварительно надо выделить форму `Form1`, выполнив щелчок мышью в любой ее свободной области или на ее заголовке.

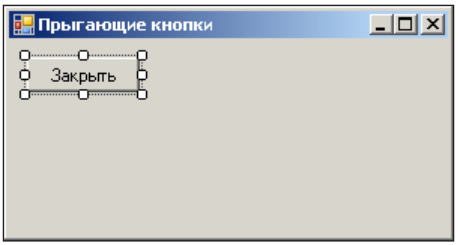


Рис. 4.1. Вид формы `Form1` для проекта EVENTS на начальном этапе разработки

Листинг 4.1. Настройка свойств

```
Form1: Text = Прыгающие кнопки, StartPosition = CenterScreen
button1: Text = Закрыть
```

Листинг 4.2. Обработчик `button1.Click`

```
private void button1_Click(object sender, EventArgs e)
{
  Close();
}
```

Листинг 4.3. Обработчик `Form1.MouseDown`

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
  button1.Location = new Point(e.X - button1.Width / 2,
    e.Y - button1.Height / 2);
}
```

Результат. После запуска программы на экране появляется окно **Прыгающие кнопки** с кнопкой **Закрыть**. Благодаря значению **`CenterScreen`**, установленному для свойства `StartPosition`, окно располагается в центре экрана. При щелчке мышью в любой точке окна кнопка **Закрыть** услужливо прыгает на указанное место. Нажатие кнопки **Закрыть** приводит к завершению программы и возврату в среду Visual C#.

Комментарии

1. Функция `Close` является методом класса `Form1`, унаследованным от класса-предка `Form`. Поскольку обработчик события `button1_Click` (листинг 4.2) также является методом класса `Form1`, перед именем метода `Close` не нужно указывать имя вызвавшего его объекта-формы.
2. В методе `Form1_MouseDown` (листинг 4.3) изменяется значение свойства `Location` *кнопки*, поэтому необходимо явно указать объект (`button1`), свойство которого надо изменить. Свойство `Location` является структурой типа `Point` и состоит из двух полей `X` и `Y` типа `int`. Для его изменения создается новый экземпляр типа `Point`, поля которого определяются с помощью полей `X` и `Y` параметра `e` (эти поля содержат координаты точки, в которой была нажата кнопка мыши). Свойства `Width` (ширина) и `Height` (высота) кнопки используются для того, чтобы отцентрировать кнопку относительно курсора мыши. Обратите внимание на ключевое слово `new`, необходимое при вызове конструкторов структур и классов.
3. Положение и размеры любого *визуального компонента* (т. е. потомка класса `Control`), в том числе и самой формы, можно определить и изменить с помощью набора свойств. За положение отвечает уже упомянутое свойство `Location`, а также свойства `Left` и `Top` типа `int`. Эти свойства (как и поля `X` и `Y` свойства `Location`) определяют координаты левого верхнего угла компонента относительно левого верхнего угла *клиентской части* формы (в клиентскую часть формы не входит ее заголовок и рамка). В случае формы эти свойства определяют координаты левого верхнего угла формы относительно левого верхнего угла экрана. За размер отвечает свойство `Size` — структура типа `Size` с полями `Width` и `Height` типа `int` — и отдельные свойства `Width` и `Height` типа `int`. Отметим также свойства `Right` и `Bottom` типа `int`, определяющие координаты правого нижнего угла визуального компонента. Все координаты указываются в пикселах.
4. Как программа хранит информацию о компонентах, размещенных в форме, и как узнает значения свойств, которые настроены с помощью окна свойств? Вся эта информация сохраняется в текстовом файле, связанном с дизайнером формы (в нашем случае этот файл имеет имя Form1.Designer.cs).

Хотя обычно необходимости в его "ручной" корректировке нет, полезно познакомиться с его содержимым, загрузив данный файл в редактор (для этого достаточно выполнить двойной щелчок мышью на имени файла в окне **Solution Explorer**). Файл Form1.Designer.cs содержит ту часть описания класса `Form1`, которая непосредственно связана с визуальным проектированием. В частности, в конце данного файла содержится список всех компонентов, размещенных в форме. В нашей программе таким компонентом является кнопка:

```
private System.Windows.Forms.Button button1;
```

Далее, если развернуть скрытый раздел описания, помеченный текстом **Windows Form Designer generated code** (Код, сгенерированный дизайнером форм), щелкнув мышью на изображении знака +, то мы увидим фрагмент программы, содержащий все настройки свойств, сделанные нами с помощью окна **Properties**, например:

```
this.button1.Text = "Закрыть";
```

Отметим, что изменение этих свойств непосредственно в тексте файла Form1.Designer.cs немедленно скажется на виде формы. Так, если изменить приведенный выше оператор на следующий

```
this.button1.Text = "Close";
```

и переключиться в режим дизайна формы (т. е. перейти на вкладку **Form1.cs [Design]**), то текст кнопки изменится на **Close**, и этот же текст будет указан рядом со свойством `Text` кнопки в окне **Properties**.

Таким образом, все действия, связанные с размещением компонентов в форме и настройкой их свойств, можно выполнять, добавляя соответствующие операторы непосредственно в текст программы (именно так делается в книгах [5] и [6]); визуальные средства, в частности, окна **Toolbox** и **Properties**, лишь ускоряют этот процесс и делают его более наглядным.

5. Как программа узнает, что метод `button1_Click` надо вызывать нажатием кнопки `button1`, а метод `Form1_MouseDown` — щелчком на форме? Одновременно с созданием заготовки для метода `button1_Click` в окне свойств рядом с текстом **Click** появляется имя метода `button1_Click` (в этом можно убедиться, вернувшись в режим дизайна формы, выделив кнопку `button1` и перейдя в окне **Properties** в режим **Events**, нажав кнопку ). Таким образом, значение события `Click` для компонента `button1` становится равным **`button1_Click`**. В файле Form1.Designer.cs соответствующее действие оформляется в виде следующего оператора:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Надо отметить, что допускается более краткая форма записи оператора подключения обработчика к событию (мы воспользуемся этой краткой формой в *разд. 4.3*).

Все *события* компонентов, отображаемые в окна свойств в режиме **Events**, по умолчанию являются пустыми, т. е. не связанными с какими-либо обработчиками. Если же некоторое событие связано с обработчиком (в нашем случае событие `Click` связано с методом `button1_Click`), то при наступления соответствующего события (в нашем случае — при нажатии кнопки) компонент вызывает тот метод-обработчик, который присоединен к нему. При этом первый параметр обработчика (`sender`) позволяет определить, какой компонент вызвал данный обработчик, а второй параметр (`e`) может содержать дополнительную информацию о событии.

6. Где находятся операторы, с которых начинается выполнение программы? Любая программа на языке C# начинается с выполнения стартового метода, который по умолчанию имеет имя `Main`. В шаблоне Windows-приложений, создаваемых в Visual C#, метод `Main` размещается в файле Program.cs. Этот файл создается автоматически, и, подобно файлу Form1.Designer.cs, связанному с дизайнером формы, как правило, не требует корректировки. Загрузив файл Program.cs в редактор, мы увидим, что метод `Main` содержит три оператора, важнейшим из которых является последний:

```
Application.Run(new Form1());
```

Данный оператор создает экземпляр главной формы приложения (типа `Form1`) и запускает цикл обработки событий, который работает до тех пор, пока не будет закрыта главная форма. При закрытии главной формы происходит выход из цикла обработки событий, и выполнение приложения завершается.

4.2. Отключение обработчика от события

Добавьте в форму еще одну кнопку (она получит имя `button2`) и сделайте ее свойство `Text` пустым, используя окно **Properties** (рис. 4.2).

В файле Form1.cs в начало описания класса `Form1` (перед конструктором `public Form1()`) добавьте описание нового поля `r`:

```
private Random r = new Random();
```

Определите обработчики событий `MouseMove` и `Click` для кнопки `button2` (листинг 4.4).

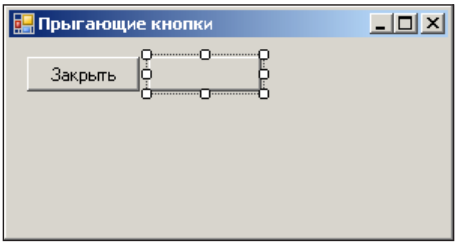


Рис. 4.2. Окончательный вид формы `Form1` для проекта EVENTS

Листинг 4.4. Обработчики `button2.MouseMove` и `button2.Click`

```
private void button2_MouseMove(object sender, MouseEventArgs e)
{
  if (Control.ModifierKeys == Keys.Control)
    return;
  // - если нажата клавиша <Ctrl>, то немедленно выйти из обработчика
  button2.Location = new Point(r.Next(ClientRectangle.Width - 5),
    r.Next(ClientRectangle.Height - 5));
}
private void button2_Click(object sender, EventArgs e)
{
  button2.Text = "Изменить";
  button2.MouseMove -= button2_MouseMove;
}
```

Результат. "Дикая" кнопка с пустым заголовком не дает на себя нажать, убегая от курсора мыши. Для того чтобы ее приручить, надо переместить на нее курсор, держа нажатой клавишу <Ctrl>. После щелчка на "дикой" кнопке она приручается: на ней появляется заголовок **Изменить** и она перестает убегать от курсора мыши. Следует заметить, что приручить кнопку можно и с помощью клавиатуры, выделив эту кнопку с помощью клавиши <Tab> (или клавиш со стрелками) и нажав на клавишу <Пробел>.

"Прирученная" кнопка пока ничего не делает. Это будет исправлено в *разд. 4.3*.

Комментарии

1. В данном разделе демонстрируется возможность *отключения* метода-обработчика от события, с которым он ранее был связан. Для этого исполь-

зуется операция -=, слева от которой указывается событие, а справа — тот обработчик, который надо отсоединить от события.

2. Для обеспечения случайного перемещения "дикой" кнопки в программе используется объект r типа Random *(датчик случайных чисел)*, позволяющий генерировать равномерно распределенные псевдослучайные числа. Для создания и инициализации объекта типа Random можно использовать два варианта конструктора: без параметров и с параметром seed типа int. Если параметр seed не указан, то датчик инициализируется значением, полученным на основе текущего времени (по часам компьютера). Объекты типа Random, инициализированные с помощью одинаковых значений seed, будут генерировать одинаковые последовательности случайных чисел.

   Для получения случайного числа типа int в классе Random предусмотрен метод Next, имеющий три варианта: без параметров (возвращает число в диапазоне от 0 до максимального значения типа int, не включая само это значение), с одним параметром max (возвращает число в диапазоне от 0 до max, не включая max) и с двумя параметрами min и max (возвращает число в диапазоне от min до max, не включая max). В обработчике button2_MouseMove (см. листинг 4.4) использован вариант метода Next с одним параметром. Имеется также метод NextDouble без параметров, возвращающий случайное число типа double, лежащее в полуинтервале [0, 1).

3. В обработчике button2_MouseMove использованы свойства формы ClientHeight и ClientWidth — высота и ширина ее клиентской части (напомним, что в клиентскую часть формы не включаются ее заголовок и рамка). Вычитание числа 5 гарантирует, что убежавшая кнопка будет видна на экране (хотя бы частично).

4. Следует обратить внимание на то, как в обработчике button2_MouseMove проверяется, нажата ли клавиша <Ctrl>. Как было сказано в *разд. 4.1*, обычно дополнительная информация о произошедшем событии передается в обработчик с помощью второго параметра e. Например, в обработчике button2_MouseMove с помощью этого параметра (типа MouseEventArgs) можно определить, в каком месте кнопки button2 находится в данный момент курсор мыши (поля e.X и e.Y типа int), а также нажата ли при этом какая-либо кнопка мыши (поле e.Button типа MouseButtons). Однако информацию о нажатых в данный момент управляющих клавишах параметр e типа MouseEventArgs не содержит. Тем не менее, подобную информацию можно получить с помощью статического свойства ModifierKeys класса Control — базового предка всех визуальных компонентов. С помощью этого свойства можно определить, нажаты ли в данный момент клавиши

<Ctrl>, <Alt>, <Shift>, а также любые их комбинации. Например, проверить, нажата ли комбинация клавиш <Ctrl>+<Shift>, можно с помощью следующего условия (скобки являются обязательными):

```
Control.ModifierKeys == (Keys.Control | Keys.Shift)
```

5. Обратите внимание на то, что поле `r` в классе `Form1` не только описывается, но и сразу инициализируется (с помощью конструктора класса `Random` без параметров). В какой момент выполняется данная инициализация? По правилам языка C# явно указанные операторы инициализации всех полей класса автоматически помещаются в начало любого конструктора класса. Таким образом, поле `r` будет инициализировано в начале выполнения конструктора формы `Form1` (перед выполнением оператора `InitializeComponent()`, указанного в теле конструктора). Разумеется, можно было бы поступить иначе: описать поле `r` типа `Random` без его инициализации

   ```
   private Random r;
   ```

   и поместить в конструктор формы `Form1` оператор

   ```
   r = new Random();
   ```

   Следует также отметить, что модификатор доступа `private` (означающий, что данное поле является *закрытым*, т. е. доступно только для методов класса `Form1`) можно не указывать, поскольку если у члена класса отсутствует модификатор доступа, этот член автоматически снабжается модификатором `private`. Тем не менее, мы всегда будем указывать модификаторы доступа, так как это делает программный код более наглядным.

4.3. Подключение к событию другого обработчика

Для того чтобы "прирученная" кнопка при нажатии на нее выполняла какие-либо действия, можно добавить требуемые действия к уже имеющемуся обработчику `button2_Click`. Однако в этом случае обработчик должен проверять, в каком состоянии находится кнопка: "диком" или "прирученном". Поступим по-другому: свяжем событие `Click` для "прирученной" кнопки с другим обработчиком. Такой подход позволит продемонстрировать ряд особенностей, связанных с действиями по подключению и отключению обработчиков.

Новый обработчик `button2_Click2` создайте "вручную", не прибегая к услугам окна свойств **Properties**. Для этого в конце описания класса `Form1` в файле Form1.cs (*перед* двумя последними закрывающими фигурными скобками `}`) добавьте описание нового обработчика (листинг 4.5). Обратите внимание на то,

что в листинге 4.5 выделены полужирным шрифтом все строки. Это означает, что необходимо набрать весь его текст. Кроме того, добавьте новые операторы в методы `button2_Click` (листинг 4.6) и `Form1_MouseDown` (листинг 4.7). Напомним, что если место добавления не уточняется, то операторы надо добавлять в *конец* метода.

Листинг 4.5. Метод `button2_Click2`

```
private void button2_Click2(object sender, EventArgs e)
{
  if (WindowState == FormWindowState.Normal)
    WindowState = FormWindowState.Maximized;
  else
    WindowState = FormWindowState.Normal;
}
```

Листинг 4.6. Добавление к методу `button2_Click`

```
button2.Click -= button2_Click;
button2.Click += button2_Click2;
```

Листинг 4.7. Добавление к методу `Form1_MouseDown`

```
if (button2.Text != "")
{
  button2.Text = "";
  button2.MouseMove += button2_MouseMove;
  button2.Click += button2_Click;
  button2.Click -= button2_Click2;
}
```

Результат. "Прирученная" кнопка теперь выполняет полезную работу. Щелчок на ней приводит к разворачиванию окна программы на весь экран, а новый щелчок восстанавливает первоначальное состояние окна. Если же щелкнуть мышью на форме (не на кнопке), то услужливая кнопка **Закрыть** прибежит на вызов, а "прирученная" кнопка **Изменить** снова одичает, потеряет текст своего заголовка и начнет убегать от мыши (см. комментарий 1).

Недочет. При выполнении программы может возникнуть ситуация, когда одна или обе кнопки не будут отображаться в форме (если, например, кнопки были перемещены на новое место при развернутом окне, после чего окно было возвращено в исходное состояние).

Исправление. Определите обработчик события `SizeChanged` для формы `Form1` (листинг 4.8).

Листинг 4.8. Обработчик `Form1.SizeChanged`

```
private void Form1_SizeChanged(object sender, EventArgs e)
{
  if (!ClientRectangle.IntersectsWith(button1.Bounds))
    button1.Location = new Point(10, 10);
  if (!ClientRectangle.IntersectsWith(button2.Bounds))
    button2.Location = new Point(10, 40);
}
```

Результат. Теперь в ситуации, когда при изменении размера формы ее кнопки оказываются вне клиентской части, происходит перемещение этих кнопок на явно указанные позиции около левого верхнего угла формы (см. комментарий 2).

Комментарии

1. Приведенные тексты методов (см. листинги 4.6 и 4.7) показывают, что при смене обработчика недостаточно подсоединить к событию новый обработчик; необходимо также отсоединить от события обработчик, ранее связанный с ним. Дело в том, что к одному и тому же событию можно последовательно присоединить несколько обработчиков (для этого достаточно применить к событию несколько раз операцию +=), хотя подобная возможность для событий визуальных компонентов используется редко (обратите внимание на то, что с помощью окна **Properties** *нельзя* подключить к одному событию несколько обработчиков). При явном подключении обработчиков к событию необходимо следить за тем, чтобы один и тот же обработчик не был присоединен к событию несколько раз, поскольку многократное подключение обработчика обычно приводит к появлению трудно выявляемых ошибок. Такую ситуацию можно проиллюстрировать с помощью нашей программы, закомментировав заголовок оператора `if` в обработчике `Form1_MouseDown`:

   ```
   // if (button2.Text != "")
   ```

Если теперь после запуска программы несколько раз щелкнуть мышью на форме, а затем приручить кнопку `button2`, то при ее последующем нажатии форма несколько раз последовательно перейдет из развернутого состояния в стандартное и обратно. Это объясняется тем, что в измененном варианте программы *каждый* щелчок на форме присоединяет к событию `Click` кнопки `button2` новый экземпляр обработчика `button2_Click`, и при нажатии на эту кнопку каждый экземпляр обработчика последовательно запускается на выполнение. Ситуация осложняется еще тем обстоятельством, что в программе невозможно выяснить, сколько и какие обработчики подсоединены в настоящий момент к событию (а не зная этого, нельзя и обеспечить отсоединение от события всех его обработчиков). Итак, к действиям по явному подключению обработчика к событию, а также его последующему отключению, следует подходить крайне внимательно.

2. Для проверки взаимного расположения формы и ее компонентов были использованы свойства типа `Rectangle`, предназначенные только для чтения: `ClientRectangle` возвращает прямоугольник, определяющий клиентскую часть формы (или визуального компонента), а `Bounds` возвращает прямоугольник, определяющий положение визуального компонента на форме (или формы на экране). Структура `Rectangle` имеет ряд свойств (в том числе `Location`, `Size`, `Left`, `Top`, `Width`, `Height`, `Right`, `Bottom`), а также несколько полезных методов. Например, метод `IntersectsWith`, использованный в листинге 4.8, позволяет проверить, является ли непустым пересечение двух прямоугольников (в нашей программе анализируется пересечение прямоугольника `ClientRectangle` формы `Form1` с прямоугольником `Bounds` для каждой из двух кнопок: `button1` и `button2`).

Глава 5

Формы: проект FORMS

Проект FORMS знакомит с особенностями приложений, использующих несколько форм, и демонстрирует различные способы настройки внешнего вида форм и режимы их отображения на экране. Рассматриваются вопросы взаимодействия форм в рамках одного приложения и, в частности, проблемы, связанные с закрытием немодальных подчиненных форм. Описываются настройки для диалоговых окон и методы, обеспечивающие вывод на экран стандартных диалогов.

5.1. Настройка визуальных свойств форм. Открытие форм в обычном и модальном режиме

После создания проекта FORMS добавьте к нему две новые формы (они автоматически получат имена `Form2` и `Form3`). Разместите в форме `Form1` две кнопки `button1` и `button2`. Настройте свойства всех форм и компонентов (листинг 5.1, рис. 5.1—5.3). В начало описания класса `Form1` добавьте описания двух полей:

```
private Form2 form2 = new Form2();
private Form3 form3 = new Form3();
```

Дополните конструктор класса `Form1` (листинг 5.2) и определите обработчики события `Shown` для формы `Form1` и события `Click` для кнопок `button1` и `button2` (листинг 5.3).

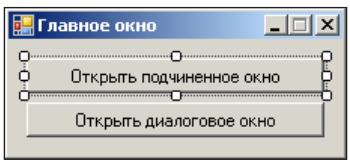


Рис. 5.1. Окончательный вид формы `Form1` для проекта FORMS

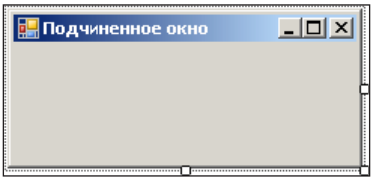


Рис. 5.2. Вид формы Form2 для проекта FORMS на начальном этапе разработки

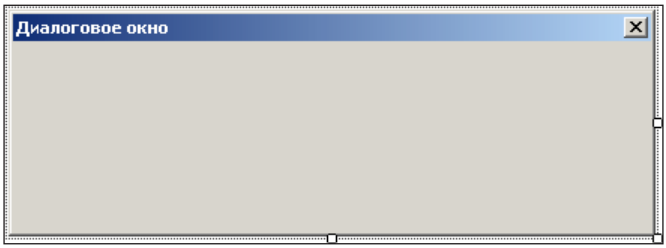


Рис. 5.3. Вид формы Form3 для проекта FORMS на начальном этапе разработки

Листинг 5.1. Настройка свойств

```
Form1: Text = Главное окно, MaximizeBox = False,
  FormBorderStyle= FixedSingle
Form2: Text = Подчиненное окно, StartPosition = Manual,
  ShowInTaskbar = False
Form3: Text = Диалоговое окно, MaximizeBox = False, MinimizeBox = False,
  FormBorderStyle = FixedDialog, StartPosition = CenterScreen,
  ShowInTaskbar = False
button1: Text = Открыть подчиненное окно
button2: Text = Открыть диалоговое окно
```

Листинг 5.2. Новый вариант конструктора формы Form1

```
public Form1()
{
  InitializeComponent();
  AddOwnedForm(form2);
  AddOwnedForm(form3);
}
```

Листинг 5.3. Обработчики `Form1.Shown`, `button1.Click` и `button2.Click`

```
private void Form1_Shown(object sender, EventArgs e)
{
  form2.Location = new Point(Right - 10, Bottom - 10);
}
private void button1_Click(object sender, EventArgs e)
{
  form2.Show();
}
private void button2_Click(object sender, EventArgs e)
{
  form3.ShowDialog();
}
```

Результат. Программа содержит три формы, демонстрирующие основные типы окон в графических Windows-приложениях: окно фиксированного размера (класс `Form1`), окно переменного размера (класс `Form2`), диалоговое окно (класс `Form3`). Форма `Form1` содержит две кнопки (см. рис. 5.1); формы `Form2` и `Form3` пока не содержат компонентов. Форма `Form1` является главной; она автоматически создается при запуске приложения и сразу отображается на экране. Кроме того, она создает две формы с именами `form2` и `form3` — экземпляры классов `Form2` и `Form3` соответственно (см. комментарий 1).

Форма `form2` (подчиненная форма) вызывается из главной формы нажатием кнопки **Открыть подчиненное окно**; при этом она отображается в обычном режиме (методом `Show`). Форма `form3` также является подчиненной; она вызывается нажатием кнопки **Открыть диалоговое окно** и отображается в *модальном (диалоговом)* режиме (методом `ShowDialog`). Особенность модального режима состоит в том, что если некоторая форма приложения находится в этом режиме, то до ее закрытия нельзя переключаться на другие формы приложения (хотя возможно переключение на другие запущенные приложения). Для завершения программы надо закрыть ее главную форму.

Главная форма `Form1` имеет фиксированные размеры. Размеры подчиненной формы `form2` можно изменять; кроме того, форму `form2` можно разворачивать на весь экран. Визуальные свойства формы `form3` соответствуют стандартным свойствам диалогового окна: размеры формы `form3` нельзя изменять и, кроме того, в ее заголовке отображается только текст и кнопка закрытия (см. рис. 5.3). См. также комментарий 2.

При открытии формы Form1 место для ее размещения выбирается операционной системой, форма form2 отображается около правого нижнего угла формы Form1 с небольшим наложением, форма form3 всегда отображается в центре экрана (см. комментарий 3).

Ошибка. После закрытия формы form2 попытка ее повторного открытия приводит к исключительной ситуации с диагностикой "Cannot access a disposed object" ("Нет доступа к разрушенному объекту"). Это связано с тем, что закрытие формы, открытой в обычном режиме, приводит к ее разрушению. Заметим, что если форма открыта в диалоговом режиме, то ее разрушения при закрытии не происходит, в чем можно убедиться, несколько раз открыв и закрыв форму form3.

Исправление. Определите обработчик события FormClosing для формы Form2 (листинг 5.4).

Листинг 5.4. Обработчик Form2.FormClosing

```
private void Form2_FormClosing(object sender, FormClosingEventArgs e)
{
  if (e.CloseReason == CloseReason.UserClosing)
  {
    e.Cancel = true;
    Hide();
  }
}
```

Результат. Теперь форму form2, как и форму form3, можно многократно закрывать и открывать в ходе выполнения программы (см. комментарий 4).

Комментарии

1. Вызов метода f1.AddOwnedForm(f2) добавляет форму f2 к списку подчиненных форм формы f1. При этом, в частности, свойство Owner формы f2 становится равным f1 (заметим, что вместо указанного вызова метода можно было бы использовать оператор присваивания вида f2.Owner = f1). Подчиненная форма всегда отображается поверх главной, даже если главная форма является активной. Кроме того, при минимизации или закрытии главной формы ее подчиненные формы также минимизируются или, соответственно, закрываются.

2. Благодаря явному заданию значения **`False`** для свойств `ShowInTaskbar` подчиненных форм, кнопки для этих форм не отображаются на панели задач в нижней части экрана. За стиль границы формы отвечает свойство `FormBorderStyle`, за отображение/скрытие элементов заголовка отвечают логические свойства `MinimizeBox` (кнопка минимизации), `MaximizeBox` (кнопка максимизации), `ShowIcon` (значок в заголовке формы), `ControlBox` (данное свойство позволяет скрыть все элементы заголовка, кроме текста) и `HelpButton` (позволяет отобразить кнопку с вопросительным знаком, но только при условии, что скрыты и кнопка минимизации, и кнопка максимизации). Стиль границы **`FixedDialog`**, в отличие от стиля **`FixedSingle`**, обеспечивает автоматическое скрытие значка в заголовке формы.

3. За начальное расположение формы на экране отвечает свойство `StartPosition`, равное по умолчанию значению **`WindowsDefaultLocation`** (при этом позиция формы определяется операционной системой). Для размещения формы по центру экрана свойство `StartPosition` следует положить равным **`CenterScreen`**. Для явной установки начальной позиции с помощью свойства `Location` формы необходимо, чтобы свойство `StartPosition` было равно **`Manual`** (в противном случае значение свойства `Location` игнорируется). Если свойство `StartPosition` не равно **`Manual`**, то свойства, связанные с положением формы (`Location`, `Left`, `Top` и т. д.) получат правильные значения только в момент первого отображения формы на экране. С первым отображением формы связывается событие `Shown`, поэтому начальное положение подчиненной формы `form2` мы определяем именно в обработчике события `Shown` для главной формы `Form1`, когда положение главной формы на экране уже известно (листинг 5.3).

4. Событие `FormClosing` относится к группе событий, которые возникают перед выполнением некоторого действия и позволяют отменить его. Второй параметр (`e`) у обработчиков подобных событий имеет изменяемое свойство `Cancel`, которому следует присвоить значение `true`, если требуется отменить соответствующее действие. В приведенном обработчике (листинг 5.4) отменяется закрытие формы `form2`; вместо этого она просто удаляется с экрана методом `Hide` (аналогичного результата можно добиться, установив значение ее свойства `Visible` равным `false`). Условие, указанное в обработчике, позволяет определить, что привело к попытке закрыть форму. Это условие будет истинным при попытке закрыть форму любым из способов, доступных пользователю программы, а также при явном вызове метода `Close`. В то же время, при автоматическом вызове метода `Close` в момент закрытия главной формы данное условие будет ложным, что позволит "по-настоящему" закрыть подчиненную форму при завершении работы приложения.

5.2. Контроль за состоянием подчиненной формы. Воздействие подчиненной формы на главную

Измените метод `button1_Click` (листинг 5.5) и определите обработчик события `VisibleChanged` для формы `Form2` (листинг 5.6).

Листинг 5.5. Новый вариант метода `button1_Click`

```
private void button1_Click(object sender, EventArgs e)
{
  form2.Visible = !form2.Visible;
}
```

Листинг 5.6. Обработчик `Form2.VisibleChanged`

```
private void Form2_VisibleChanged(object sender, EventArgs e)
{
  Owner.Controls["button1"].Text = Visible ?
    "Закрыть подчиненное окно" : "Открыть подчиненное окно" ;
}
```

Результат. Теперь текст кнопки `button1` и действия при ее нажатии зависят от того, отображается на экране подчиненное окно `form2` или нет: если подчиненное окно присутствует на экране, то оно исчезает, а если его на экране нет, то оно появляется. Подчиненное окно можно закрыть не только с помощью кнопки `button1`, но и любым стандартным способом, принятым в Windows (например, с помощью комбинации клавиш <Alt>+<F4>); при любом способе закрытия подчиненного окна текст кнопки `button1` будет изменен.

Комментарии

1. В то время как главная форма для доступа к подчиненной может просто обратиться к ней по имени, подчиненная форма так сделать не может, поскольку имя главной формы ей неизвестно (главная форма Windows-приложения вообще не имеет имени, поскольку она создается "на лету", как параметр метода `Application.Run` — см. комментарий 6 в *разд. 4.1*). Однако подчиненная форма может обратиться к главной форме, используя свое свойство `Owner`. Более того, с помощью свойства-коллекции `Controls`

(типа `ControlCollection`), имеющегося у любой формы (а точнее, у любого компонента-*контейнера*), подчиненная форма может получить доступ ко всем компонентам своего владельца. Для обращения к элементам коллекции `Controls` можно использовать либо числовые индексы, либо *строковые ключи* — имена компонентов. Так, в листинге 5.6 вместо строкового ключа `button1` можно было бы указать число `1`, поскольку компоненты формы нумеруются в порядке, *обратном* их размещению в форме: в нашем случае кнопка `button2` (размещенная в форме последней) имеет минимальный индекс `0`, а кнопка `button1` — индекс `1`. Заметим, что следствием такого способа нумерации компонентов является то, что размещение в форме нового компонента приводит к изменению индексов *всех* ее компонентов. По этой причине предпочтительнее использовать не числовые индексы, а строковые ключи, соответствующие именам требуемых компонентов. К вопросам, связанным с порядком размещения компонентов в форме, мы еще вернемся в примере MOUSE *(см. разд. 10.1)*.

2. В методе `Form2_VisibleChanged` (см. листинг 5.6) использована тернарная операция

   условие `?` *выражение1* `:` *выражение2*

   Если *условие* истинно, то вычисляется и возвращается *выражение1*, а если *условие* ложно, то вычисляется и возвращается *выражение2*. Подчеркнем, что при выполнении тернарной операции вычисляется только то выражение, значение которого будет возвращено. Мы использовали тернарную операцию, так как она приводит к более компактному коду, чем эквивалентный ей вариант с полным условным оператором `if-then-else`:

```
if (Visible)
  Owner.Controls["button1"].Text = "Закрыть подчиненное окно";
else
  Owner.Controls["button1"].Text = "Открыть подчиненное окно";
```

5.3. Компоненты, подстраивающиеся под размер окна

Разместите в форме `Form2` компонент-*метку* типа `Label` (метка получит имя `label1`) и настройте ее свойства (листинг 5.7; при задании свойств `Dock` и `TextAlign` на экране можно отобразить вспомогательную графическую панель, в которой достаточно щелкнуть на центральном прямоугольном элементе). В результате форма `Form2` примет вид, приведенный на рис. 5.4.

В начало описания класса Form2 добавьте описание поля count:

```
private int count;
```

Заметим, что при создании формы ее поле count будет автоматически инициализировано нулевым значением.

В метод Form2_VisibleChanged добавьте новые операторы (листинг 5.8).

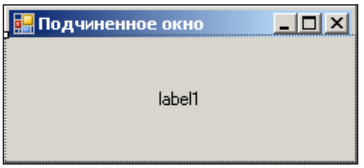


Рис. 5.4. Окончательный вид формы Form2 для проекта FORMS

Листинг 5.7. Настройка свойств

```
label1: AutoSize = False, Dock = Fill, TextAlign = MiddleCenter
```

Листинг 5.8. Новый вариант метода Form2_VisibleChanged

```
private void Form2_VisibleChanged(object sender, EventArgs e)
{
  Owner.Controls["button1"].Text = Visible ?
    "Закрыть подчиненное окно" : "Открыть подчиненное окно" ;
  if (Visible)
    label1.Text = "Окно открыто в " + (++count) + "-й раз.";
}
```

Результат. При изменении размеров подчиненного окна Form2 размеры находящейся на нем метки label1 изменяются так, чтобы она занимала всю внутреннюю *(клиентскую)* часть окна. Текст метки содержит информацию о том, сколько раз было открыто подчиненное окно.

Комментарии

1. При использовании *операции инкремента* вида ++i (префиксный вариант операции) вначале происходит увеличение значения переменной i на 1, а затем данная переменная используется в выражении. Для постфиксной операции i++ действия выполняются в обратном порядке: вначале прежнее значение i используется в выражении, а затем это значение увеличивается

на 1. Аналогичным образом ведут себя префиксный и постфиксный варианты *операции декремента* `--`.

2. Обратите внимание на то, что для преобразования числового значения `++count` в его строковое представление не требуется вызывать метод `ToString`, поскольку, согласно правилам языка C#, если один из операндов операции + является строкой, то для другого операнда автоматически вызывается метод `ToString`. Заметим, что при формировании строк из нескольких элементов вместо операции + можно также использовать метод `Format` класса `string` (см. комментарий 1 в *разд. 10.1*).

5.4. Модальные и обычные кнопки диалогового окна

Разместите в форме `Form3` две метки (`label1` и `label2`), два компонента типа `TextBox` *(поля ввода)*, которые получат имена `textBox1` и `textBox2`, а также три кнопки (`button1`, `button2` и `button3`). Настройте свойства формы `Form3` и добавленных компонентов (листинг 5.9). При настройке взаимного расположения компонентов (рис. 5.5) следует начать с полей ввода `TextBox`, после чего выровнять по этим полям связанные с ними метки.

Определите обработчик события `Click` для кнопки `button2`, размещенной в форме `Form3` (листинг 5.10), и измените метод `button2_Click` формы `Form1` (листинг 5.11).

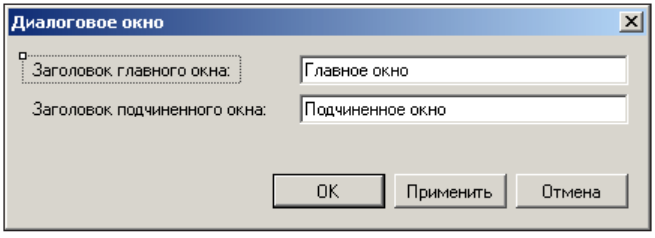


Рис. 5.5. Окончательный вид формы `Form3` для проекта FORMS

Листинг 5.9. Настройка свойств

```
Form3: AcceptButton = button1, CancelButton = button3
label1: Text = Заголовок главного окна:
label2: Text = Заголовок подчиненного окна:
textBox1: Text = Главное окно
textBox2: Text = Подчиненное окно
```

```
button1: Text = OK, DialogResult = OK
button2: Text = Применить
button3: Text = Отмена
```

Листинг 5.10. Обработчик button2.Click (для кнопки формы Form3)

```
internal void button2_Click(object sender, EventArgs e)
// модификатор доступа изменен на internal
{
  Owner.Text = textBox1.Text;
  Owner.OwnedForms[0].Text = textBox2.Text;
}
```

Листинг 5.11. Новый вариант метода button2_Click формы Form1

```
private void button2_Click(object sender, EventArgs e)
{
  if (form3.ShowDialog() == DialogResult.OK)
  // - проверка, по нажатию какой кнопки было закрыто диалоговое окно
    form3.button2_Click(this, EventArgs.Empty);
}
```

Результат. Диалоговое окно Form3 позволяет изменить заголовки главного и подчиненного окна. Заголовки окон изменяются либо при нажатии обычной кнопки **Применить**, либо при нажатии *модальной* кнопки **OK** (в последнем случае диалоговое окно закрывается). Окно также закрывается при нажатии модальной кнопки **Отмена**; в этом случае заголовки окон не изменяются. Вместо кнопки **OK** можно нажать клавишу <Enter>, вместо кнопки **Отмена** — клавишу <Esc>.

Комментарии

1. Для того чтобы нажатие на кнопку приводило к закрытию диалогового окна, кнопку необходимо сделать модальной, установив значение ее свойства DialogResult отличным от **None** (значение по умолчанию). Заметим, что при установке свойства CancelButton для формы указанная в этом свойстве кнопка (в нашем случае **button3**) автоматически получает значение DialogResult, равное **Cancel**. Сама форма также имеет свойство

DialogResult; если форма открыта в диалоговом режиме, то присваивание ее свойству DialogResult значения, отличного от **None**, приводит к немедленному закрытию формы, причем полученное значение DialogResult возвращается функцией ShowDialog, отобразившей форму на экране. При нажатии на модальную кнопку значение ее свойства DialogResult просто присваивается одноименному свойству формы. При открытии формы в обычном (немодальном) режиме описанный выше механизм не действует.

2. Для доступа из формы Form3 к свойству Text формы Form2 используется тот факт, что эти формы имеют общего *владельца* (owner), который хранит список своих подчиненных форм (в порядке их подключения) в свойстве-массиве OwnedForms типа Form[]. В отличие от свойства-коллекции Controls *(см. разд. 5.2)*, свойство OwnedForms, будучи обычным массивом, допускает только целочисленное индексирование.

3. Явный вызов метода button2_Click класса Form3 в обработчике button2_Click класса Form1 обеспечивает выполнение действий, связанных с нажатием на кнопку **Применить** (таким образом, этот вызов *имитирует* нажатие на кнопку). При вызове этого метода в качестве параметров традиционно указываются this, т. е. сама форма Form1 (см. описание термина "this" в *приложении 1*) и EventArgs.Empty. Вместо этого можно было бы дважды указать константу null, поскольку параметры в методе button2_Click класса Form3 не используются. В нашей книге мы будем в дальнейшем использовать null вместо EventArgs.Empty.

   Для возможности вызова метода button2_Click класса Form3 из класса Form1 модификатор доступа для этого метода необходимо изменить на internal (модификатор internal обеспечивает свободный доступ к члену класса *в пределах создаваемого проекта*). Можно было бы указать и модификатор public, обеспечивающий свободный доступ к члену класса даже из других проектов, при условии подключения к ним данного проекта. Обычно это делается при разработке проектов, являющихся *библиотеками классов*; в результате компиляции подобных проектов создаются не исполняемые файлы, а файлы с расширением dll.

5.5. Установка активного компонента формы

Определите обработчик события VisibleChanged для формы Form3 (листинг 5.12).

Листинг 5.12. Обработчик `Form3.VisibleChanged`

```
private void Form3_VisibleChanged(object sender, EventArgs e)
{
  if (Visible)
    ActiveControl = textBox1;
}
```

Результат. Независимо от того, какой компонент диалогового окна был активным в момент его закрытия, при следующем открытии окна всегда оказывается активным поле ввода `textBox1`. Таким образом, диалоговое окно всегда отображается в одном и том же начальном состоянии. Подобное поведение желательно обеспечивать для любых диалоговых окон.

5.6. Запрос на подтверждение закрытия формы

Измените метод `Form2_FormClosing` (листинг 5.13).

Листинг 5.13. Новый вариант метода `Form2_FormClosing`

```
private void Form2_FormClosing(object sender, FormClosingEventArgs e)
{
  if (e.CloseReason == CloseReason.UserClosing)
  {
    e.Cancel = true;
    if (MessageBox.Show("Закрыть подчиненное окно?", "Подтверждение",
      MessageBoxButtons.YesNo, MessageBoxIcon.Question,
      MessageBoxDefaultButton.Button2) == DialogResult.Yes)
      Hide();
  }
}
```

Результат. Перед закрытием подчиненного окна `form2` одним из способов, предусмотренных в системе Windows, в стандартном диалоговом окне **Подтверждение** выводится запрос на подтверждение закрытия (рис. 5.6). При выборе варианта **Нет** (который предлагается по умолчанию) закрытие подчи-

ненного окна отменяется. При закрытии главного окна открытое подчиненное окно закрывается без запроса.

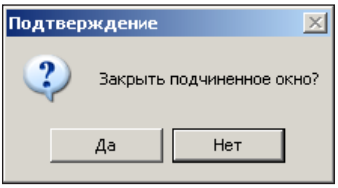


Рис. 5.6. Диалоговое окно **Подтверждение**

Недочет 1. При выборе в диалоговом окне варианта **Да** подчиненное окно закрывается, но главное окно не становится активным.

Это объясняется тем обстоятельством, что владельцем диалогового окна `MessageBox` является та форма, которая в данный момент была активной на экране (в нашем случае активной формой является `form2`), и именно эта форма должна активизироваться при закрытии окна `MessageBox`. Однако при выборе варианта **Да** форма `form2` закрывается, и поэтому ее активизация оказывается невозможной. В подобной ситуации ни одно окно на экране не будет активным, а главное окно нашей программы, скорее всего, будет заслонено окном среды Visual C#.

Исправление. В методе `Form2_FormClosing` замените оператор `Hide()` на следующий составной оператор:

```
{
  Hide();
  Owner.Activate();
}
```

> ***ПРИМЕЧАНИЕ***
>
> Другим вариантом исправления недочета 1 является явное указание *владельца* окна `MessageBox` в дополнительном параметре, который должен располагаться первым в списке параметров. Например, в качестве этого параметра можно указать `Owner`. В этом случае при выборе варианта **Да** будет успешно активизирована главная форма. Однако эта же форма будет активизирована и при выборе варианта **Нет** (когда подчиненная форма останется на экране), что представляется не вполне естественным.

Недочет 2. При закрытии подчиненного окна с помощью нажатия кнопки `button1` главного окна запрос на подтверждение закрытия не выводится.

Это происходит потому, что при выполнении обработчика `button1_Click` не вызывается метод `Close` подчиненной формы (форма просто изменяет свой ре-

жим видимости), а следовательно, не выполняется и обработчик, связанный с методом Close подчиненной формы.

Исправление. Измените метод button1_Click формы Form1 (листинг 5.14).

Листинг 5.14. Новый вариант метода button1_Click формы Form1

```
private void button1_Click(object sender, EventArgs e)
{
  if (form2.Visible)
    form2.Close();
  else
    form2.Show();
}
```

Комментарии

1. В методе Form2_FormClosing (листинг 5.13) использован один из наиболее полных вариантов метода Show класса MessageBox, позволяющий указать текст запроса, заголовок окна запроса, набор кнопок для данного окна, значок в окне и кнопку, предлагаемую по умолчанию. Любой параметр, кроме первого, может отсутствовать; при этом должны отсутствовать и все следующие за ним параметры. Если отсутствует второй параметр, то заголовок окна является пустым, если третий, то в окне отображается единственная кнопка **OK**, если четвертый — значок в окне не отображается, если пятый — кнопкой по умолчанию является первая кнопка.
2. В библиотеке .NET имеется также метод, позволяющий вывести на экран диалоговое окно, предназначенное для ввода строковой информации: это метод InputBox класса Interaction. Следует, однако, иметь в виду, что класс Interaction определен в пространстве имен Microsoft.VisualBasic, а соответствующая библиотека не подключается автоматически к проектам на языке C#. Для подключения этой библиотеки надо выполнить следующие действия: щелкнуть правой кнопкой мыши на строке **References** в окне **Solution Explorer**, выбрать из контекстного меню команду **Add Reference...** и в появившемся окне **Add Reference** выделить пункт **Microsoft.VisualBasic**, после чего нажать кнопку **OK**. Для возможности использования краткого имени класса Interaction (без указания его пространства имен) в начало программы надо добавить оператор

   ```
   using Microsoft.VisualBasic;
   ```

Метод `InputBox` имеет пять обязательных параметров: `Prompt` (строка-приглашение), `Title` (строка-заголовок), `DefaultResponse` (строка — вариант ответа по умолчанию), `XPos` и `YPos` (экранные координаты левого верхнего угла окна). Для центрирования диалогового окна по горизонтали и/или вертикали соответствующий параметр (`XPos` и/или `YPos`) надо положить равным `-1`. Метод возвращает введенную строку, если диалоговое окно было закрыто кнопкой **OK**, или пустую строку, если для закрытия окна была использована кнопка **Cancel**.

К сожалению, метод `InputBox` имеет один недостаток: кнопка отмены всегда содержит английский текст **Cancel**, что не соответствует стилю других диалоговых окон, в которых для надписей на кнопках применяется язык операционной системы (например, для русской версии Windows используются русские надписи: **Отмена**, **Да**, **Нет** и т. д.; исключением является надпись **OK**).

Мы воспользуемся методом `InputBox` в *главе 32 (см. разд. 32.6)*.

Глава 6

Совместное использование обработчиков событий и работа с клавиатурой: проект CALC

Проект CALC знакомит с совместным использованием обработчиков событий несколькими компонентами. Кроме того, в нем демонстрируется использование метода `TryParse` для обработки ошибок ввода числовых данных и рассматриваются различные варианты ускорения работы с помощью клавиатуры (кнопки по умолчанию, клавиши-ускорители и использование события `KeyPress`).

6.1. Обработчик событий для нескольких компонентов

После создания проекта CALC разместите в форме `Form1` два поля ввода (`textBox1` и `textBox2`), две метки (`label1` и `label2`) и пять кнопок (`button1`, ..., `button5`). Для того чтобы уменьшить размер первых четырех кнопок одинаковым образом (рис. 6.1), следует после размещения этих кнопок в форме выделить их (например, охватив пунктирной рамкой) и изменить размер одной из них; при этом размер остальных выделенных кнопок изменится автоматически.

Настройте свойства формы и всех добавленных компонентов (листинг 6.1).

Определите обработчик события `Click` для кнопки `button1` (листинг 6.2) и свяжите полученный обработчик `button1_Click` с событием `Click` кнопок `button2`, `button3` и `button4`. Для этого надо выбрать в окне свойств **Properties** режим **Events**, для каждой из этих кнопок перейти на строку с событием **Click** и выбрать имя обработчика `button1_Click` из выпадающего списка (выполнять двойной щелчок на поле ввода *не следует*). Можно выполнить связывание сразу для всех трех кнопок; для этого надо предварительно выделить все эти кнопки.

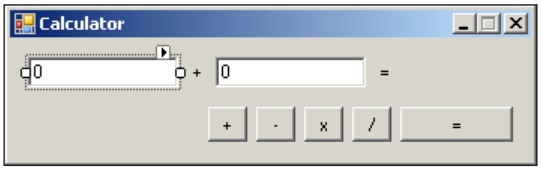


Рис. 6.1. Вид формы `Form1` для проекта CALC на начальном этапе разработки

Листинг 6.1. Настройка свойств

```
Form1: Text = Calculator, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
textBox1: Text = 0
textBox2: Text = 0
label1: Text = +
label2: Text = =
button1: Text = +
button2: Text = -
button3: Text = x
button4: Text = /
button5: Text = =
```

Листинг 6.2. Обработчик `button1.Click`

```
private void button1_Click(object sender, EventArgs e)
{
  label1.Text = (sender as Button).Text;
}
```

Результат. Нажатие на любую кнопку с обозначением операции (+, –, **x**, /) приводит к отображению этой операции в метке `label1` между полями ввода `textBox1` и `textBox2`.

Подчеркнем, что для всех четырех кнопок был задан один *общий* обработчик. Это возможно, благодаря использованию в методе-обработчике `button1_Click` параметра `sender`, в котором передается компонент, вызвавший обработчик. Операция `as` преобразует параметр `sender` (типа `object`) к типу `Button` (если этого не сделать, то произойдет ошибка компиляции, поскольку класс `object` не имеет свойства `Text`).

6.2. Вычисления с контролем правильности исходных данных

Определите обработчик события `Click` для кнопки `button5` (листинг 6.3).

Листинг 6.3. Обработчик `button5.Click`

```
private void button5_Click(object sender, EventArgs e)
{
  double x = 0,
    x1 = double.Parse(textBox1.Text),
    x2 = double.Parse(textBox2.Text);
  switch (label1.Text[0])
  {
    case '+':
      x = x1 + x2; break;
    case '-':
      x = x1 - x2; break;
    case 'x':
      x = x1 * x2; break;
    case '/':
      x = x1 / x2; break;
  }
  label2.Text = "= " + x;
}
```

Результат. При нажатии кнопки = выражение, указанное в окне программы, вычисляется и отображается в метке `label2`. В качестве второго операнда при любой операции можно указывать число 0; при делении на 0 результат имеет вид **–бесконечность** или **бесконечность** (в зависимости от знака первого ненулевого операнда). Если оба операнда равны 0, то результат имеет значение **NaN** ("не число"). По поводу особых значений типа `double` см. комментарий 2 в *разд. 3.1*. См. также комментарий 1.

Недочет. Если одно из полей ввода не содержит текст или этот текст нельзя преобразовать в число (например, **abc**), то при нажатии кнопки = возникает исключение `FormatException`. Если программа запущена в режиме с включенным отладчиком (режим **Debug**, запуск клавишей <F5>), то ее выполнение будет прервано, а в редакторе среды Visual C# будет подсвечен оператор,

вызвавший ошибку. Возможные действия в этой ситуации подробно описаны в *разд. 3.1*.

Исправление. Измените метод `button5_Click` (листинг 6.4).

Листинг 6.4. Новый вариант метода `button5_Click`

```
private void button5_Click(object sender, EventArgs e)
{
  double x = 0, x1, x2;
  if (!double.TryParse(textBox1.Text, out x1) ||
    !double.TryParse(textBox2.Text, out x2))
  {
    label2.Text = "= ERROR";
    return;
  }
  switch (label1.Text[0])
  {
    case '+':
      x = x1 + x2; break;
    case '-':
      x = x1 - x2; break;
    case 'x':
      x = x1 * x2; break;
    case '/':
      x = x1 / x2; break;
  }
  label2.Text = "= " + x;
}
```

Результат. Теперь при попытке вычислить выражение с недопустимыми операндами в метке `label2` выводится текст **ERROR**; при этом прерывание выполнения программы не происходит, и окно с сообщением об ошибке не возникает (см. комментарий 2).

Комментарии

1. Для преобразования строк в вещественные числа `x1` и `x2` в первом варианте метода `button5_Click` (см. листинг 6.3) использовался метод `Parse` для типа `double` (см. также комментарий 1 в *разд. 3.1*). Для обратного преобра-

зования полученного числа x в его строковое представление в нашем случае не надо вызывать метод ToString, поскольку переменная x используется в выражении "= " + x, при вычислении которого автоматически производится требуемое преобразование (см. комментарий 2 в *разд. 5.3*).

2. При исправлении отмеченного недочета (см. листинг 6.4) мы воспользовались методом TryParse, который, в отличие от метода Parse, никогда не приводит к возбуждению исключения. Метод TryParse для типа double возвращает true, если строка содержит правильное представление вещественного числа, и false в противном случае. Результат преобразования строки в вещественное число возвращается во втором, выходном параметре. Обратите внимание на то, что в языке C# выходные параметры при вызове метода должны снабжаться специальным модификатором out.

6.3. Простейшие приемы ускорения работы с помощью клавиатуры

Настройте свойства формы и компонентов-кнопок (листинг 6.5; в результате форма примет вид, приведенный на рис. 6.2).

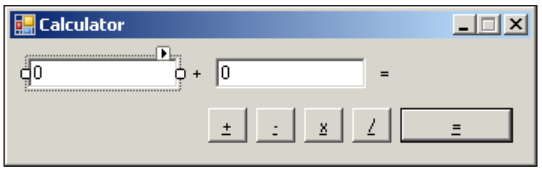


Рис. 6.2. Окончательный вид формы Form1 для проекта CALC

Листинг 6.5. Настройка свойств

```
Form1: AcceptButton = button5
button1: Text = &+
button2: Text = &–
button3: Text = &x
button4: Text = &/
button5: Text = &=
```

Результат. Благодаря установке свойства AcceptButton формы Form1 равным **button5**, кнопка = становится *кнопкой по умолчанию*, эквивалентом ее нажатия будет нажатие на клавишу <Enter> (кнопка по умолчанию обводится более толстой рамкой). Символы, указанные на кнопках, подчеркиваются; это

является признаком того, что с каждой кнопкой связана клавиша-ускоритель <Alt>+<*подчеркнутый символ*>.

> ***ПРИМЕЧАНИЕ***
>
> Возможна ситуация, когда после запуска программы символы, связанные с клавишами-ускорителями, не будут подчеркнуты. В этом случае следует нажать клавишу <Alt>.

Ошибка. После нажатия на любую кнопку с арифметической операцией все последующие вычисления возвращают значение 0 (поскольку первым символом метки `label1` теперь является символ `&`, не предусмотренный в операторе `switch`).

Исправление. Измените оператор в методе `button1_Click` (см. листинг 6.2) следующим образом:

```
label1.Text = (sender as Button).Text[1].ToString();
```

Это исправление приводит к тому, что в метку копируется только второй символ заголовка кнопки (т. е. символ с индексом `1`). Поскольку в C# запрещено присваивать символьное выражение строковой переменной, полученный символ необходимо преобразовать к типу `string` с помощью метода `ToString`.

6.4. Использование обработчика событий от клавиатуры

С помощью окна **Properties** установите значение свойства `KeyPreview` формы `Form1` равным **`True`**. Определите обработчик события `KeyPress` для формы `Form1` (листинг 6.6).

Листинг 6.6. Обработчик `Form1.KeyPress`

```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
  char c = e.KeyChar;
  switch (c)
  {
    case '+':
      button1_Click(button1, null); break;
    case '_':
```

```
      button1_Click(button2, null); break;
    case 'x':
    case '*':
      button1_Click(button3, null); break;
    case '/':
      button1_Click(button4, null); break;
    case '=':
      button5_Click(button5, null); break;
  }
  e.Handled = !(char.IsDigit(c) || c == ',' || c == '-' || c == '\b');
}
```

Результат. Теперь для ввода любой операции достаточно нажать соответствующую клавишу (поскольку клавиша <–> используется при вводе отрицательных чисел, в качестве ускорителя для кнопки `button2` выбрана комбинация клавиш <Shift>+<–>, соответствующая символу подчеркивания "_"). Для ввода операции умножения можно использовать как клавишу <x>, так и клавишу <*>. При вводе чисел игнорируются все клавиши, кроме цифровых, <–>, <,> и <Backspace> (для обозначения символа, генерируемого клавишей <Backspace>, в C# можно использовать управляющую последовательность `\b`; нажатие этой клавиши обеспечивает удаление символа, расположенного *слева* от курсора в активном поле ввода). См. также комментарии 1 и 2.

Недочет. В обработчике `Form1_KeyPress` предполагается, что десятичным разделителем является *запятая*, тогда как в системе Windows при других региональных настройках может использоваться разделитель-*точка*.

Исправление. Замените в методе `Form1_KeyPress` фрагмент условия

```
c == ','
```

на следующий:

```
c == Application.CurrentCulture.NumberFormat.NumberDecimalSeparator[0]
```

Результат. Теперь в качестве допустимого символа используется десятичный разделитель, соответствующий текущим региональным настройкам системы Windows. Для тестирования этой возможности достаточно временно изменить региональные настройки в разделе **Язык и региональные стандарты** Панели управления Windows, выбрав, например, вариант языка **Английский (США)**. См. также комментарий 3.

Комментарии

1. Для того чтобы события от клавиатуры вначале обрабатывались формой, а затем — активным компонентом, свойство формы `KeyPreview` необходимо положить равным **`True`**. Если этого не сделать, то событие от клавиатуры может быть сразу обработано активным компонентом и, таким образом, *не дойдет* до формы и, соответственно, ее клавиатурного обработчика.
2. Если в обработчике события `KeyPress` установить для параметра `e` (типа `KeyPressEventArgs`) свойство `Handled` равным `true`, то текущее событие от клавиатуры будет считаться обработанным и не будет передано в другие компоненты, находящиеся "в цепочке активности". Так, в нашем случае форма перехватывает и обрабатывает все символы, кроме цифровых символов, десятичного разделителя, минуса и `\b`.
3. Класс `Application` имеет свойство `CurrentCulture` типа `CultureInfo`, определенного в пространстве имен `System.Globalization`. Это свойство позволяет получить информацию о *региональных настройках* (т. е. о числовых и денежных форматах, а также форматах для указания даты и времени), которые используются программой. По умолчанию программа использует региональные настройки операционной системы. Необходимость в указании индекса `[0]` для свойства `NumberDecimalSeparator` обусловлено тем, что это свойство имеет *строковый* тип (`string`), который в языке C# нельзя сравнивать с *символьным* типом (`char`). Свойство `NumberDecimalSeparator` доступно только для чтения, однако в программе можно изменить свойство `CurrentCulture` в целом (см. по этому поводу комментарий в *разд. 12.1*).

6.5. Контроль за изменением исходных данных

Измените метод `button1_Click` (листинг 6.7), определите обработчик события `TextChanged` для компонента `textBox1` (листинг 6.8) и свяжите полученный обработчик `textBox1_TextChanged` с событием `TextChanged` компонента `textBox2`.

Листинг 6.7. Новый вариант метода `button1_Click`

```
private void button1_Click(object sender, EventArgs e)
{
  label1.Text = (sender as Button).Text[1].ToString();
  label2.Text = "=";
}
```

Листинг 6.8. Обработчик `textBox1.TextChanged`

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
  label2.Text = "=";
}
```

Результат. При изменении операции или содержимого текстовых полей результат предыдущего вычисления стирается. Это действие предотвращает возможность рассогласования между исходными данными и полученным результатом, из-за которого в прежних вариантах программы на экране мог появиться, например, текст **2 + 2 = 5**.

Глава 7

Курсоры и значки: проект CURSORS

Проект CURSORS знакомит с особенностями использования в приложении курсоров (классы `Cursor` и `Cursors`) и значков (класс `Icon`). Кроме того, в нем приводятся начальные сведения о классах-обобщениях (generics) и о внедрении в приложение графических ресурсов. Также рассматривается компонент `NotifyIcon`, предназначенный для работы с областью уведомлений (traybar) панели задач.

7.1. Использование стандартных курсоров

После создания проекта CURSORS разместите в форме `Form1` кнопку `button1` и настройте свойства формы и добавленной кнопки (листинг 7.1; рис. 7.1).

В начало описания класса `Form1` добавьте описания двух полей:

```
private List<string> str = new List<string>(30);
private List<Cursor> cur = new List<Cursor>(30);
```

Добавьте новые операторы в конструктор класса `Form1` (листинг 7.2) и определите обработчик события `MouseDown` для кнопки `button1` (листинг 7.3).

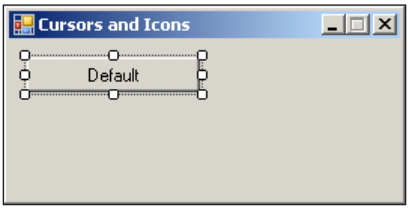


Рис. 7.1. Вид формы `Form1` для проекта CURSORS на начальном этапе разработки

Листинг 7.1. Настройка свойств

```
Form1: Text = Cursors and Icons, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
button1: Text = Default
```

Листинг 7.2. Новый вариант конструктора формы Form1

```
public Form1()
{
  InitializeComponent();
  foreach (System.Reflection.PropertyInfo pi in
    typeof(Cursors).GetProperties())
  {
    str.Add(pi.Name);
    cur.Add((Cursor)pi.GetValue(null, null));
  }
  button1.Tag = str.IndexOf("Default");
}
```

Листинг 7.3. Обработчик button1.MouseDown

```
private void button1_MouseDown(object sender, MouseEventArgs e)
{
  int k = (int)button1.Tag,
    c = str.Count;
  switch (e.Button)
  {
    case MouseButtons.Left:
      k = (k + 1) % c; break;
    case MouseButtons.Right:
      k = (k - 1 + c) % c; break;
  }
  button1.Text = str[k];
  button1.Cursor = cur[k];
  button1.Tag = k;
}
```

Результат. Нажатие мышью на кнопку **Default** приводит к смене курсора для данной кнопки; при этом заголовок кнопки заменяется на имя текущего курсора. Перебор всех 28 стандартных курсоров выполняется циклически; порядок перебора курсоров соответствует порядку их размещения в выпадающем списке для свойства `Cursor` в окне **Properties**. При нажатии не левой, а правой кнопки мыши курсоры перебираются в обратном порядке.

Комментарии

1. Все стандартные курсоры можно получить с помощью класса `Cursors`, в котором для каждого стандартного курсора предусмотрено особое свойство, доступное только для чтения. В классе `Cursors` отсутствуют средства для циклического перебора стандартных курсоров, а также для получения их имен. Поэтому в конструкторе формы (см. листинг 7.2) мы формируем две коллекции типа класса-обобщения `List<T>`: строковую коллекцию `str`, содержащую имена всех стандартных курсоров, и коллекцию `cur`, содержащую сами эти курсоры (т. е. объекты типа `Cursor`) в том же порядке. При формировании коллекций используется *механизм отражения* (reflection), применяемый к классу `Cursors`. Механизм отражения в нашей книге подробно не рассматривается (ознакомиться с ним можно, например, в книгах [3], глава 14; [7], глава 20, [8], глава 22); отметим лишь, что с его помощью определяется массив объектов типа `PropertyInfo`, каждый из которых содержит информацию о каком-либо *свойстве* класса `Cursors`, а затем в цикле `foreach` выполняется циклический перебор этих объектов и извлечение из них сведений об именах и значениях соответствующих свойств класса `Cursors`.

2. С помощью *классов-обобщений* (generics), появившихся в .NET 2.0, можно легко создавать строго типизированные коллекции, в частности, динамические массивы `List<T>`, определенные в пространстве имен `System.Collections.Generic`. При описании коллекций типа `List<T>` необходимо сразу указывать тип `T` их элементов. В нашей программе коллекция `str` описана как `List<string>`, т. е. предназначена для хранения элементов-строк, а коллекция `cur` описана как `List<Cursor>` и предназначена для хранения элементов типа `Cursor`.

   В ранних версиях .NET можно было использовать лишь коллекции, элементами которых считались объекты базового типа `object` (примером такой коллекции является класс `ArrayList`, определенный в пространстве имен `System.Collections`). Применение подобных коллекций было потенциально опасным, поскольку ошибки, связанные с несоответствием типов данных, помещаемых в коллекцию, можно было выявить только на этапе выполнения приложения. Кроме того, пользоваться такими коллекциями было неудобно, так как при обращении к их элементам требовалось явно преобразовывать элементы к нужному типу. Коллекции-обобщения со строгой типизацией более надежны (в них нельзя записать данные, тип которых отличается от типа элементов коллекции, поскольку ошибки подобного рода выявляются уже на этапе компиляции) и более удобны в использовании (при доступе к элементам коллекции нет необходимости выполнять преобразование типа). Подробному рассмотрению обобщений посвящена глава 16 книги [8]; см. также главу 4 в книге [3].

3. Индекс курсора, связанного с кнопкой `button1`, мы сохраняем в свойстве `Tag` этой кнопки. Свойство `Tag` имеется у любого визуального компонента; оно может использоваться для хранения какой-либо его дополнительной характеристики. Поскольку свойство `Tag` имеет тип `object`, ему можно присвоить значение любого типа, однако при *считывании* его значения необходимо выполнять явное преобразование к фактическому типу содержащегося в нем объекта (в нашем случае к типу `int`). Последний оператор в конструкторе формы (см. листинг 7.2) присваивает свойству `Tag` кнопки `button1` индекс того элемента коллекции `cur`, который соответствует стандартному курсору `Default`.
4. В методе `button1_MouseDown` (см. листинг 7.3) циклический перебор индексов в диапазоне от `0` до `c` (где `c = str.Count – 1`, а `str.Count` равно количеству элементов, содержащихся в коллекции `str`) выполняется с применением операции `%` (взятие остатка от деления нацело). Дополнительное слагаемое `c` в выражении `(k – 1 + c) % c` добавлено для того, чтобы выражение в скобках никогда не принимало отрицательных значений.
5. Для обработки нажатий на кнопку `button1` было использовано событие `MouseDown`, так как событие `Click` компонента `Button` реагирует только на *левую* кнопку мыши (для других компонентов ситуация может быть иной, например событие `Click` компонента `Label` реагирует на щелчок любой кнопкой мыши — *см. разд. 12.3*).

7.2. Установка курсора для формы и индикация режима ожидания с помощью курсора

Разместите в форме `Form1` три новые кнопки: `button2`, `button3`, `button4` и присвойте свойствам `Text` этих кнопок значения **`Form Cursor`**, **`Wait Cursor`** и **`Default Form Cursor`** соответственно (рис. 7.2). Определите обработчики события `Click` для добавленных кнопок (листинг 7.4).

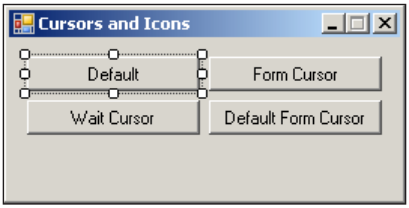


Рис. 7.2. Вид формы `Form1` для проекта CURSORS на промежуточном этапе разработки

Листинг 7.4. Обработчики `button2.Click`, `button3.Click` и `button4.Click`

```
private void button2_Click(object sender, EventArgs e)
{
  Cursor = button1.Cursor;
}
private void button3_Click(object sender, EventArgs e)
{
  UseWaitCursor = true;
}
private void button4_Click(object sender, EventArgs e)
{
  UseWaitCursor = false;
  Cursor = Cursors.Default;
}
```

Результат. Нажатие кнопки **Form Cursor** распространяет действие курсора, связанного с первой кнопкой (`button1`), на всю форму, включая находящиеся на ней кнопки. Нажатие кнопки **Wait Cursor** устанавливает для всей формы и ее компонентов *курсор ожидания* (wait cursor) в виде песочных часов. Нажатие кнопки **Default Form Cursor** отменяет курсор ожидания и восстанавливает стандартный курсор для формы (но не для первой кнопки, курсор которой по-прежнему соответствует названию, приведенному в ее заголовке).

Комментарии

1. Если свойство `Cursor` визуального компонента равно `null` или имеет значение `Cursors.Default`, то вид курсора для этого компонента определяется его родительским компонентом (в нашем случае формой); для формы вид курсора со значением `Default` определяется операционной системой. Именно поэтому при изменении курсора для формы автоматически изменяется курсор для кнопок `button2`, `button3` и `button4`, у которых свойство `Cursor` равно `Default`.
2. Логическое свойство `UseWaitCursor` имеется у любого визуального компонента. Если свойство `UseWaitCursor` какого-либо компонента положить равным `true`, то текущий курсор этого компонента и всех его дочерних компонентов заменится на курсор ожидания (песочные часы).

7.3. Подключение к проекту новых курсоров и их сохранение в виде внедренных ресурсов

Кроме стандартных курсоров в программе можно использовать дополнительные курсоры. Файлы с курсорами имеют расширение cur; их можно найти, например, в подкаталоге Cursors каталога Windows. Следует отметить, что *анимированные* курсоры (файлы с расширением ani) использовать в Visual C# нельзя, а *цветные* курсоры при использовании в программе отображаются в монохромном режиме. Значок любого cur-файла соответствует изображению курсора, содержащегося в этом файле; это позволяет быстро ознакомиться с содержимым cur-файлов, используя приложения Проводник или Мой компьютер.

Выберите два каких-либо cur-файла, содержащих монохромные курсоры, и скопируйте их в каталог проекта CURSORS под именами C1.cur и C2.cur.

Загрузите в среду Visual C# проект CURSORS, если это еще не сделано.

Выполните команду меню **Project | Add Existing Item...**. В появившемся диалоговом окне введите имя файла C1.cur (это можно сделать непосредственно в строке ввода; можно также указать в выпадающем списке **Files of type** вариант **All Files** и выбрать файл C1.cur из списка всех файлов, содержащихся в каталоге). Нажмите клавишу <Enter> или кнопку **Add**. В результате файл C1.cur будет добавлен к проекту CURSORS, в чем можно убедиться, посмотрев на окно **Solution Explorer.**

Выделите файл C1.cur в окне **Solution Explorer**; при этом в окне **Properties** будут отображены его свойства. Установите свойство `Build Action` равным **`Embedded Resource`**; прочие свойства изменять не требуется.

Аналогичными действиями добавьте к проекту файл C2.cur и настройте его свойство `Build Action`.

В конструктор класса `Form1` добавьте новые операторы (листинг 7.5).

Листинг 7.5. Добавление к конструктору формы `Form1`

```
for (int i = 1; i <= 2; i++)
{
  str.Add("C" + i);
  cur.Add(new Cursor(GetType(), "C" + i + ".cur"));
}
```

Результат. При создании формы новые курсоры загружаются в программу и добавляются к списку доступных курсоров под именами `C1` и `C2` . В дальнейшем они обрабатываются аналогично стандартным *(см. разд. 7.1 и 7.2)*. Следует подчеркнуть, что изображения этих курсоров размещаются непосредственно в исполняемом файле приложения CURSORS.exe.

Комментарии

1. Для загрузки курсора из ресурсов приложения используется вариант конструктора класса `Cursor` с двумя параметрами, первый из которых обычно имеет вид `GetType()`, а второй задает имя соответствующего ресурса. Можно также загрузить курсор непосредственно из cur-файла; для этого предназначен конструктор класса `Cursor` с одним параметром — именем файла (если cur-файл содержится в рабочем каталоге приложения, то путь к файлу можно не указывать).
2. Среда Visual Studio имеет встроенные средства, позволяющие создавать новые курсоры и редактировать имеющиеся. Для создания нового курсора (и его немедленного подключения к текущему проекту) достаточно выполнить команду **Project | Add New Item...**, в появившемся окне указать тип нового элемента (**Cursor file**) и его имя, после чего нажать клавишу <Enter> или кнопку **Add**. Будет создана заготовка для монохромного курсора с указанным именем, которая немедленно загрузится в редактор среды.

   Редактировать изображение курсора можно с помощью различных инструментов, кнопки которых отображаются на соответствующей панели. Для выбора цвета линий и фона используется окно **Colors** (цвет линий выбирается с помощью левой кнопки мыши, а фона — с помощью правой). Кроме черно-белых шаблонов, имитирующих оттенки серого цвета, можно выбирать два особых цвета: зеленый (соответствует прозрачной части курсора) и розовый (соответствует той части курсора, под которой будет происходить инвертирование цветов).

   Для созданного курсора необходимо указать *точку привязки* (hot spot), т. е. пиксел, определяющий положение курсора на экране (например, в случае стандартного курсора-стрелки его точкой привязки является острие стрелки). Точка привязки определяется с помощью кнопки **Set Hot Spot Tool**  на панели инструментов рисования. Кроме того, координаты точки привязки указываются (и могут быть изменены) в окне свойств курсора.

   Для редактирования существующего курсора, подключенного к проекту, достаточно выполнить двойной щелчок мышью на его имени в окне **Solution Explorer**; в результате выбранный курсор будет загружен в редактор среды.

Экспресс-вариант Visual C# не имеет встроенных средств для создания и редактирования изображений (в том числе курсоров).

7.4. Работа со значками

Небольшие изображения, называемые *значками*, или *пиктограммами*, хранятся в файлах с расширением ico. Большое количество ico-файлов содержится в библиотеке изображений, входящей в состав системы Visual Studio 2005 (библиотека поставляется в виде архива VS2005ImageLibrary.zip, который размещается в подкаталоге Common7\VS2005ImageLibrary каталога Visual Studio). Аналогичная библиотека с именем VS2008ImageLibrary входит в состав системы Visual Studio 2008. В экспресс-варианты Visual C# библиотека изображений не входит, однако найти несколько ico-файлов на компьютере не составляет труда. Значок ico-файла, подобно значку cur-файла, соответствует содержащемуся в нем изображению.

Добавление к проекту существующих ico-файлов выполняется аналогично добавлению cur-файлов. При описании последующих действий предполагается, что к проекту добавлены файлы Computer.ico и Folder.ico (в библиотеке VS2005ImageLibrary они находятся в подкаталоге icons\Misc, а в библиотеке VS2008ImageLibrary — в подкаталоге Objects\ico_format\Office & Dev). Как и в случае cur-файлов, после добавления ico-файла в проект следует установить его свойство `Biuld Action` равным **`Embedded Resource`.**

Разместите в форме кнопку `button5` и присвойте ее свойству `Text` значение **`Icon 0`**. В начало описания класса `Form1` добавьте описание массива `ico`:

```
private Icon[] ico = new Icon[3];
```

Добавьте в конструктор класса `Form1` новые операторы (листинг 7.6) и определите обработчик события `Click` для кнопки `button5` (листинг 7.7).

Листинг 7.6. Добавление к конструктору формы `Form1`

```
button5.Tag = 0;
ico[1] = new Icon(GetType(), "Computer.ico");
ico[2] = new Icon(GetType(), "Folder.ico");
ico[0] = Icon;
```

Листинг 7.7. Обработчик `button5.Click`

```
private void button5_Click(object sender, EventArgs e)
{
  int k = ((int)button5.Tag + 1) % 3;
```

```
  button5.Text = "Icon " + k;
  button5.Tag = k;
  Icon = ico[k];
}
```

Результат. Кнопка `button5` позволяет изменить значок формы как в ее заголовке, так и на ее кнопке, находящейся на панели задач (в нижней части экрана). Значок с номером 0 соответствует исходному значку формы, присвоенному ей по умолчанию. Для хранения трех значков в программе используется массив `ico` фиксированного размера.

Комментарии

1. Листинги 7.6 и 7.7 показывают, что для работы со значками применяются те же приемы, что и для работы с курсорами; требуется лишь использовать класс `Icon` и свойство `Icon` формы. Загрузить значок можно и непосредственно из ico-файла; для этого в классе `Icon` предусмотрен конструктор с одним параметром — именем файла.
2. Значок может быть связан не только с формой, но и с приложением в целом, однако значок приложения используется только при отображении соответствующего exe-файла в Проводнике или подобных ему файловых браузерах, поддерживающих отображение значков. Для задания значка приложения надо открыть в редакторе среды Visual C# вкладку со свойствами проекта, используя команду меню **Project | *<имя проекта>* Properties...**, и указать нужный значок в строке **Icon**, находящейся в разделе **Application**.

7.5. Размещение значка в области уведомлений

Разместите в форме еще одну кнопку (`button6`), а также компонент `NotifyIcon` (этот компонент получит имя `notifyIcon1`), и настройте их свойства (листинг 7.8, рис. 7.3).

Заметим, что компонент `NotifyIcon` не является визуальным компонентом, поэтому при его добавлении в форму он размещается в специальной *области невизуальных компонентов*, расположенной под изображением формы.

В конструкторе класса `Form1` дополните последний оператор следующим образом:

```
notifyIcon1.Icon = ico[0] = Icon;
```

В методе `button5_Click` также дополните последний оператор:

```
notifyIcon1.Icon = Icon = ico[k];
```

Определите обработчик события `Click` для кнопки `button6` (листинг 7.9).

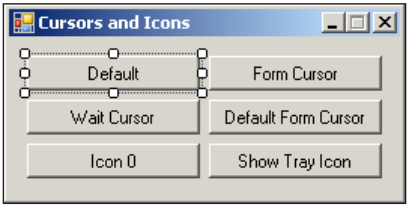


Рис. 7.3. Окончательный вид формы `Form1` для проекта CURSORS

Листинг 7.8. Настройка свойств

```
button6: Text = Show Tray Icon
notifyIcon1: Text = Icon in Traybar, Visible = False
```

Листинг 7.9. Обработчик `button6.Click`

```
private void button6_Click(object sender, EventArgs e)
{
  bool b = button6.Text == "Show Tray Icon";
  notifyIcon1.Visible = b;
  ShowInTaskbar = !b;
  button6.Text = b ? "Hide Tray Icon" : "Show Tray Icon";
}
```

Результат. Нажатие кнопки `button6` скрывает кнопку формы на панели задач и отображает ее значок в *области уведомлений* (notification area, traybar) в правой части панели задач. При наведении курсора на этот значок возникает всплывающая подсказка "Icon in Traybar". Повторное нажатие кнопки `button6` восстанавливает исходное представление формы на панели задач.

Комментарии

1. Операторы, измененные в конструкторе класса `Form1` и в методе `button5_Click`, имеют вид *`переменная1 = переменная2 = выражение`*. Подобные операторы позволяют одновременно присвоить значение выражения всем указанным переменным. Эта возможность связана с тем, что операция при-

сваивания вида `a = b` не только присваивает значение выражения `b` переменной `a`, но и *возвращает* присвоенное значение. Заметим, что при наличии нескольких операций присваивания они выполняются справа налево, т. е. в записи `a = b = c` скобки неявно расставляются следующим образом: `a = (b = c)`.

2. Обычно в области уведомлений размещаются значки приложений, работающих в фоновом режиме и использующих окна только для просмотра и изменения своих настроек. Кроме всплывающих подсказок, возникающих при наведении курсора на значок, компонент `NotifyIcon` позволяет выводить на экран рядом со значком сообщения в "пузыре" (используя метод `ShowBalloonTip` и связанные с ним свойства `BalloonTipIcon`, `BalloonTipTitle`, `BalloonTipText`), а также связывать со значком *контекстное меню*, вызываемое с помощью правой кнопки мыши (свойство `ContextMenuStrip`; работа с контекстными меню описывается в *разд. 20.3*). Значок в области уведомлений может также реагировать на события от мыши; с этой целью в компоненте `NotifyIcon` предусмотрен ряд событий, в том числе `Click` и `DblClick`.

Глава 8

Поля ввода: проект TEXTBOXES

Проект TEXTBOXES демонстрирует особенности работы с полями ввода (компонентами `TextBox`). Рассматриваются вопросы, связанные с активизацией компонента (получением и потерей фокуса ввода) и порядком обхода компонентов, а также с обработкой ошибочных данных на уровне отдельного поля ввода и формы в целом. Описывается механизм генерации сообщений об ошибках, основанный на использовании компонента `ErrorProvider`.

8.1. Дополнительное выделение активного поля ввода

После создания проекта TEXTBOXES разместите в форме `Form1` 12 полей ввода `TextBox` (`textBox1`—`textBox12`) и настройте свойства формы и добавленных компонентов (листинг 8.1). Компоненты `TextBox` следует размещать в форме по строкам в направлении слева направо: `textBox1`—`textBox3` в первом ряду, `textBox4`—`textBox6` во втором ряду и т. д. (рис. 8.1). Для того чтобы присвоить свойству `Text` всех компонентов одно и то же значение **`Data`**, достаточно выделить все компоненты и затем указать это значение с помощью окна **Properties**.

Определите обработчики событий `Enter` и `Leave` для поля ввода `textBox1` (листинг 8.2), после чего свяжите созданные обработчики с событиями `Enter` и `Leave` всех полей ввода (о связывании обработчиков с несколькими компонентами *см. в разд. 6.1*). Чтобы связать созданные обработчики одновременно со всеми оставшимися полями ввода (`textBox2`—`textBox12`), следует предварительно их выделить.

Листинг 8.1. Настройка свойств

```
Form1: Text = TextBoxes, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
textBox1-textBox12: Text = Data
```

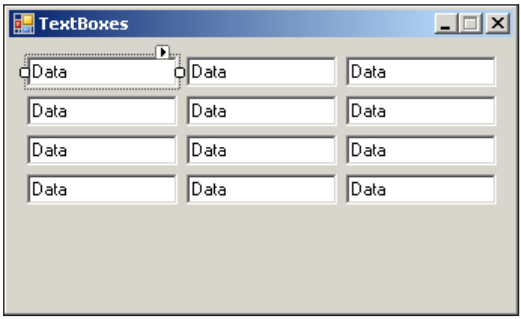


Рис. 8.1. Вид формы `Form1` для проекта TEXTBOXES на начальном этапе разработки

Листинг 8.2. Обработчики `textBox1.Enter` и `textBox1.Leave`

```
private void textBox1_Enter(object sender, EventArgs e)
{
  TextBox tb = sender as TextBox;
  tb.ForeColor = Color.White;
  tb.BackColor = Color.Green;
}
private void textBox1_Leave(object sender, EventArgs e)
{
  TextBox tb = sender as TextBox;
  tb.ForeColor = SystemColors.WindowText;
  tb.BackColor = SystemColors.Window;
}
```

Результат. При получении фокуса любым полем ввода (т. е. при активизации поля ввода — см. комментарий 1) изменяется его фон и цвет символов; при потере фокуса восстанавливается исходная цветовая настройка.

Недочет. При получении фокуса текст поля ввода автоматически выделяется (как правило, он изображается белым цветом на синем фоне); таким образом, левая часть поля ввода (выделенные символы) закрашивается синим цветом, а правая — зеленым, что выглядит некрасиво.

Исправление. Добавьте новые операторы в конструктор класса `Form1` (листинг 8.3).

Листинг 8.3. Новый вариант конструктора формы Form1

```
public Form1()
{
  InitializeComponent();
  for (int i = 1; i <= 12; i++)
  {
    TextBox tb = Controls["textBox" + i] as TextBox;
    tb.Select(tb.Text.Length, 0);
  }
}
```

Результат. Теперь при получении фокуса текст в поле ввода не выделяется, причем клавиатурный курсор (*каретка*, caret), имеющий вид вертикальной линии, располагается за последним символом текста (см. комментарий 2).

ПРИМЕЧАНИЕ

Для перебора всех полей ввода, размещенных в форме, использовалось свойство-коллекция `Controls` формы (см. комментарий 1 в *разд. 5.2*).

Комментарии

1. Перемещение фокуса обеспечивается щелчком мыши на требуемом компоненте или клавишами <Tab> и <Shift>+<Tab>. При использовании клавиш <Tab> и <Shift>+<Tab> порядок обхода тех компонентов, которые могут получать фокус, определяется значением их свойства `TabIndex`; по умолчанию порядок обхода совпадает с порядком добавления компонентов в форму. Для просмотра и изменения текущего порядка обхода можно воспользоваться командой меню **View | Tab Order...** в режиме дизайна формы. При выполнении этой команды около каждого компонента формы изображается число, равное значению его свойства `TabIndex`; для задания нового порядка обхода компонентов надо выполнять щелчок мышью на компонентах в требуемом порядке (при этом числа около компонентов изменяются). Для выхода из режима настройки порядка обхода достаточно нажать клавишу <Esc>. Заметим, что порядок обхода можно изменять и программным способом *(см. разд. 8.2)*. С обходом компонентов связано также свойство `TabStop`: если значение свойства `TabStop` некоторого визуального компонента равно **`False`**, то при обходе с помощью клавиш <Tab> и <Shift>+<Tab> этот компонент пропускается.

   Для перемещения фокуса часто бывает достаточно использовать клавиши со стрелками, однако в случае компонентов `TextBox` это невозможно, поскольку в них нажатия на клавиши со стрелками обрабатываются особым образом.

2. Для определения и изменения выделенного фрагмента текста в поле ввода предусмотрен ряд свойств и методов. В листинге 8.3 используется метод `Select`, имеющий два параметра: позицию начала выделения (нумеруется от нуля, который соответствует номеру позиции перед первым символом) и длину выделения, т. е. количество выделенных символов. Если длина выделения равна 0, то позиция начала выделения определяет позицию каретки. Для позиции начала выделения и длины выделения предусмотрены также свойства `SelectionStart` и `SelectionLength`, которые доступны и для чтения, и для записи. Так, в листинге 8.3 вместо последнего оператора можно использовать оператор `tb.SelectionStart = tb.Text.Length` (при этом свойство `SelectionLength` изменять не требуется).

   Упомянем также важное свойство `SelectedText`, которое позволяет получить и изменить выделенный текст: присваивание свойству `SelectedText` новой строки приводит к тому, что выделенный фрагмент заменяется указанной строкой (если поле ввода не содержало выделения, то указанная строка вставляется в позицию каретки). В результате изменения свойства `SelectedText` выделение с текста снимается, а каретка размещается за вставленным фрагментом текста (таким образом, после присваивания свойству `SelectedText` любого значения это свойство будет возвращать пустую строку). Если же присвоить свойству `SelectedText` пустую строку, то выделенный в поле ввода фрагмент будет удален.

   Еще одним свойством, связанным с выделением, является логическое свойство `HideSelection`. Если оно имеет значение **`True`** (являющееся значением по умолчанию), то при потере полем ввода фокуса выделенный фрагмент текста перестает отображаться другим цветом (однако при повторном получении фокуса цвет выделения восстанавливается). Если свойство `HideSelection` имеет значение **`False`**, то вид выделенного фрагмента при потере фокуса не изменяется (подобный режим обычно используется в текстовых редакторах при выполнении действий по поиску и замене фрагментов текста с применением вспомогательных диалоговых окон).

8.2. Управление порядком обхода полей в форме

Разместите в форме `Form1` компонент-*контейнер* `groupBox1` (типа `GroupBox`). После этого разместите в созданном компоненте-контейнере два компонента-*радиокнопки* (типа `RadioButton`): `radioButton1` и `radioButton2`. Настройте свойства добавленных компонентов (листинг 8.4) и их положение (рис. 8.2).

Определите обработчик события CheckedChanged для радиокнопки radioButton1 (листинг 8.5), после чего свяжите созданный обработчик с событием CheckedChanged радиокнопки radioButton2.

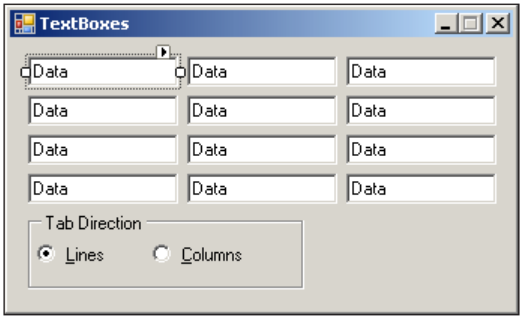


Рис. 8.2. Вид формы Form1 для проекта TEXTBOXES на промежуточном этапе разработки

Листинг 8.4. Настройка свойств

```
groupBox1: Text = Tab Direction
radioButton1: Text = &Lines, Checked = True
radioButton2: Text = &Columns
```

Листинг 8.5. Обработчик radioButton1.CheckedChanged

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
  if (!(sender as RadioButton).Checked)
    return;
  if (sender == radioButton1)
    for (int i = 0; i <= 11; i++)
      Controls["textBox" + (i + 1)].TabIndex = i;
  else
    for (int i = 0; i <= 3; i++)
      for (int j = 0; j <= 2; j++)
        Controls["textBox" + (3 * i + j + 1)].TabIndex = i + 4 * j;
}
```

Результат. С помощью радиокнопок можно изменять порядок обхода полей ввода по нажатию клавиш <Tab> и <Shift>+<Tab>: поля теперь можно обходить либо по строкам (при выбранной радиокнопке **Lines**), либо по столбцам

(при выбранной радиокнопке **Columns**). Переключать порядок обхода можно также с помощью клавиатуры, используя комбинации клавиш <Alt>+<L> и <Alt>+<C>.

Недочет. При любом из реализованных способов изменения порядка обхода полей текущее поле ввода теряет фокус (поскольку фокус принимает одна из радиокнопок).

Исправление. Измените свойства `Text` радиокнопок следующим образом: **`&Lines (F2)`** для `radioButton1` и **`&Columns (F3)`** для `radioButton2`, присвойте свойству `KeyPreview` формы `Form1` значение **`True`** и определите обработчик события `KeyDown` для формы `Form1` (листинг 8.6).

Листинг 8.6. Обработчик `Form1.KeyDown`

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
  switch (e.KeyCode)
  {
    case Keys.F2:
      radioButton1.Checked = true; break;
    case Keys.F3:
      radioButton2.Checked = true; break;
  }
}
```

Результат. Теперь для настройки порядка обхода полей по строкам достаточно нажать клавишу <F2>, а по столбцам — клавишу <F3>, причем текущее поле ввода сохраняет фокус.

Комментарий

При выборе новой радиокнопки обработчик события `CheckedChanged` запускается дважды: для ранее выбранной радиокнопки, свойство `Checked` которой изменилось с `true` на `false`, и для новой выбранной радиокнопки, свойство `Checked` которой изменилось с `false` на `true`. Первый условный оператор обработчика `radioButton1_CheckedChanged` (листинг 8.5) обеспечивает немедленный выход из него, если обработчик вызван для радиокнопки, *потерявшей* выделение. Второй условный оператор позволяет определить, какая радиокнопка стала выбранной. Заметим, что событие `CheckedChanged` наступает не только при щелчке мышью на одной из радиокнопок, но и при *программном*

изменении свойства Checked (в этом состоит отличие события CheckedChanged радиокнопки от ее же события Click, которое наступает только в результате действий пользователя).

8.3. Блокировка выхода из незаполненного поля ввода

Определите обработчик события Validating для поля ввода textBox1 (листинг 8.7), после чего свяжите созданный обработчик с событиями Validating всех полей ввода.

Листинг 8.7. Обработчик textBox1.Validating

```
private void textBox1_Validating(object sender, CancelEventArgs e)
{
  e.Cancel = (sender as TextBox).Text.Trim() == "";
}
```

Результат. Если активное поле ввода является пустым или содержит только пробелы, то выйти из него невозможно (в частности, невозможно закрыть форму). Заметим, что при этом сохраняется возможность выбора радиокнопок клавишами <F2> и <F3>, так как эта возможность не связана с потерей фокуса для активного поля ввода.

Недочет. Причина, по которой происходит блокировка фокуса, может быть непонятна пользователю. Этот недочет будет исправлен в следующем разделе.

Комментарии

1. Событие Validating возникает перед тем, как компонент потеряет фокус, причем в обработчике данного события потерю фокуса можно заблокировать, если присвоить свойству Cancel параметра e значение true (сравните это с действиями обработчика события FormClosing, рассмотренного в *разд. 5.1*).
2. Метод Trim класса string удаляет все начальные и конечные пробелы и возвращает измененную строку. Имеются также методы TrimStart и TrimEnd, которые удаляют только начальные или только конечные пробелы соответственно. С помощью методов Trim, TrimStart и TrimEnd можно удалять не только пробелы, но и любые другие символы; для этого достаточно указать символы, которые требуется удалить, в качестве параметров этих методов (число параметров может быть произвольным).

8.4. Информирование пользователя о возникшей ошибке

Добавьте к форме невизуальный компонент `ErrorProvider` (он получит имя `errorProvider1`) и положите его свойство `BlinkStyle` равным **`NeverBlink`**.

Определите обработчик события `TextChanged` для поля ввода `textBox1` (листинг 8.8), после чего свяжите созданный обработчик с событиями `TextChanged` всех полей ввода.

Листинг 8.8. Обработчик `textBox1.TextChanged`

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
  TextBox tb = sender as TextBox;
  if (tb.Text.Trim() == "")
    errorProvider1.SetError(tb, "Text must be non-empty");
  else
    if (errorProvider1.GetError(tb) != "")
      errorProvider1.SetError(tb, "");
}
```

Результат. Если в активном поле ввода удалить все непробельные символы, то справа от него появится значок в виде красного кружка с восклицательным знаком — признак сделанной ошибки. Если навести курсор мыши на этот значок, то появится всплывающая подсказка с кратким объяснением причины ошибки. При вводе в поле хотя бы одного непробельного символа значок исчезнет.

Комментарий

С помощью единственного экземпляра компонента `ErrorProvider` можно обеспечить информирование пользователя об ошибках, связанных с различными визуальными компонентами. Можно изменить значок, отображаемый на экране в случае ошибки (свойство `Icon`), а также установить один из режимов мерцания этого значка с помощью свойства `BlinkStyle` (**`NeverBlink`** — мерцания нет, **`AlwaysBlink`** — постоянное мерцание, **`BlinkIfDifferentError`** — кратковременное мерцание в момент отображения значка на экране, а также при изменении для данного компонента текста с сообщением об ошибке).

Частота мерцания (в миллисекундах) может быть настроена с помощью свойства `BlinkRate`.

8.5. Предоставление дополнительной информации об ошибке

Краткое сообщение об ошибке для некоторых пользователей может оказаться непонятным. В подобной ситуации желательно предусмотреть возможность вызова справочной системы, однако этот вызов, как правило, выполняется в помощью кнопки **Help**, в то время как выход из ошибочного поля ввода в программе заблокирован. Для решения этой проблемы предусмотрено свойство `CausesValidation`.

Разместите в форме кнопку `button1`, присвойте ее свойству `Text` значение **`Help`** (рис. 8.3), а свойству `CausesValidation` значение **`False`** и определите обработчик события `Click` для этой кнопки (листинг 8.9).

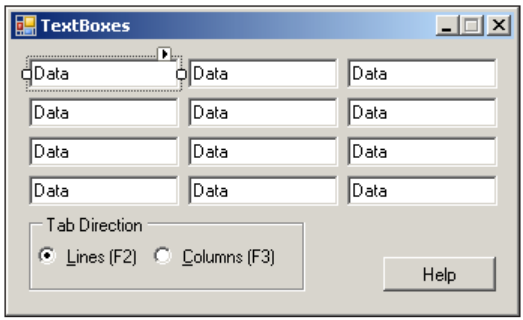


Рис. 8.3. Окончательный вид формы `Form1` для проекта TEXTBOXES

Листинг 8.9. Обработчик `button1.Click`

```
private void button1_Click(object sender, EventArgs e)
{
  MessageBox.Show("Текст в поле ввода не должен быть пустым.", "Help");
}
```

Результат. Кнопку **Help** можно нажать мышью и в ситуации, когда одно из полей ввода заблокировано. Однако в этой ситуации по-прежнему нельзя перейти на другие компоненты формы (можно лишь вернуться на поле ввода, помеченное как ошибочное).

Комментарий

Для всех визуальных компонентов свойство `CausesValidation` по умолчанию равно **`True`**. Изменять его на **`False`** рекомендуется только для кнопок, связанных с вызовом справки.

8.6. Проверка ошибок на уровне формы

Блокировка ошибочного поля ввода может оказаться излишне жесткой мерой для пользователей, которые предпочитают вначале заполнить поля, не вызывающие вопросов, а затем вернуться к тем полям, заполнение которых требует дополнительных размышлений. Для обеспечения такого способа ввода данных приложение должно в случае ошибки разрешать перемещение по компонентам, но блокировать действия, выполняемые "на уровне формы, т. е. на уровне всего набора данных как единого целого (примером таких действия является сохранение всего набора данных в файле или пересылка этого набора по сети). Реализуем подобный вариант проверки ошибок для нашего примера.

Выделите все компоненты `TextBox` и очистите для них событие `Validating` в окне свойств **Properties**, удалите описание метода `textBox1_Validating` из файла Form1.cs и определите обработчик события `FormClosing` для формы `Form1` (листинг 8.10).

Листинг 8.10. Обработчик `Form1.FormClosing`

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
  if (e.CloseReason != CloseReason.UserClosing)
    return;
  for (int i = 1; i <= 12; i++)
    if (errorProvider1.GetError(Controls["textBox" + i]) != "")
    {
      e.Cancel = true;
      return;
    }
}
```

Результат. Теперь наличие пустого поля ввода не препятствует переходу в другие поля, однако около каждого пустого поля ввода изображается значок ошибки. При наличии хотя бы одного значка ошибки форму нельзя закрыть.

Комментарий

Первый условный оператор в обработчике `Form1_FormClosing` (см. листинг 8.9) гарантирует, что форма будет закрыта, если соответствующая команда поступит не от пользователя, а от операционной системы (например, форма обязательно закроется при завершении работы Windows). См. также комментарий 4 в *разд. 5.1*.

Глава 9

Цвета: проект COLORS

Проект COLORS знакомит с типами, предназначенными для работы с цветом (`Color`, `KnownColor`, `ColorTranslator`), и с компонентами, обеспечивающими прокрутку данных (`TrackBar`, `HScrollBar`, `VScrollBar`). Кроме того, описывается способ доступа к компонентам через клавиши-ускорители связанных с ними меток, и рассматриваются варианты привязки компонентов к границам формы (anchoring).

9.1. Определение цвета как комбинации четырех цветовых составляющих. Ползунки

После создания проекта COLORS разместите в форме `Form1` компонент типа `TrackBar` (он получит имя `trackBar1`) и настройте свойства формы и компонента `trackBar1` (листинг 9.1; указанный в листинге компонент `label6` будет размещен в форме позднее).

Не снимая выделения с компонента `trackBar1`, нажмите клавиатурную комбинацию <Ctrl>+<C>, а затем четыре раза — комбинацию <Ctrl>+<V>. В результате в форму будут добавлены еще четыре компонента `TrackBar` с именами `trackBar2`, ..., `trackBar5`, причем для всех этих компонентов значения свойств будут совпадать со значениями соответствующих свойств компонента `trackBar1` (заметим, что подобное копирование компонентов можно выполнять и с помощью контекстного меню формы). Расположите все компоненты сверху вниз (рис. 9.1).

Для компонента `trackBar1` дополнительно установите свойство `Value` равным **`255`** (у остальных компонентов `TrackBar` это свойство должно остаться равным **`0`**).

После этого разместите в форме пять меток (`label1`, ..., `label5`) и установите их свойства `Text` равными соответственно **`Alpha`**, **`Red`**, **`Green`**, **`Blue`**, **`Gray`**.

Метки надо выровнять по середине соответствующих компонентов TrackBar (см. рис. 9.1). Для этого удобно использовать дополнительное средство: *панель выравнивания* **Layout**, которую можно отобразить на экране с помощью команды меню **View | Toolbars | Layout**. Для выполнения выравнивания следует *вначале* выделить один из компонентов TrackBar, а *затем*, при нажатой клавише <Shift> или <Ctrl>, — соответствующую ему метку, после чего нажать кнопку **Align Middles** на панели выравнивания. Порядок выделения компонентов важен, поскольку выравнивание производится по *текущему* компоненту, который надо выделить первым (маркеры текущего компонента имеют белый цвет).

Наконец, разместите в нижней части формы компонент-контейнер типа Panel (он получит имя panel1) и в этот компонент поместите еще одну метку (label6). Настройте свойства метки label6 (листинг 9.1; значения свойств ForeColor и BackColor можно набрать на клавиатуре, не используя вспомогательные окна выбора цвета).

Определите обработчик события Scroll для компонента trackBar1 (листинг 9.2), после чего свяжите созданный обработчик с событиями Scroll компонентов trackBar2—trackBar4 (обработчик события Scroll для компонента trackBar5 будет добавлен позднее, в *разд. 9.3*).

Для компонента panel1 задайте фоновый рисунок, взяв его из набора фоновых рисунков Windows. С этой целью вначале выделите компонент panel1 в форме (поскольку компонент panel1 находится под меткой label6, проще всего выделить эту метку, после чего нажать клавишу <Esc>). Затем выделите в окне **Properties** свойство BackgroundImage компонента panel1 и нажмите кнопку с многоточием. В появившемся окне **Select Resource** надо нажать кнопку **Import** (если кнопка недоступна, то предварительно надо выбрать переключатель **Project resource file**) и в новом окне **Open** выбрать файл с требуемым рисунком, после чего нажать клавишу <Enter> или кнопку **Открыть**. Будем считать, что выбран файл Штукатурка.bmp из каталога Windows. После выполнения описанных действий имя выбранного файла появится в списке ресурсов и будет выделено; кроме того, в правой части окна **Select Resource** появится изображение, взятое из этого файла. Осталось закрыть окно **Select Resource**, нажав кнопку **OK**. В результате свойство BackgroundImage панели panel1 получит значение **COLORS.Properties.Resources.Штукатурка**. Кроме того, в окне проекта **Solution Explorer** появится новый элемент: файл Штукатурка.bmp, размещенный в разделе **Resources**. Следует заметить, что рисунок панели будет заслонен меткой label6, поэтому увидеть его мы пока не сможем.

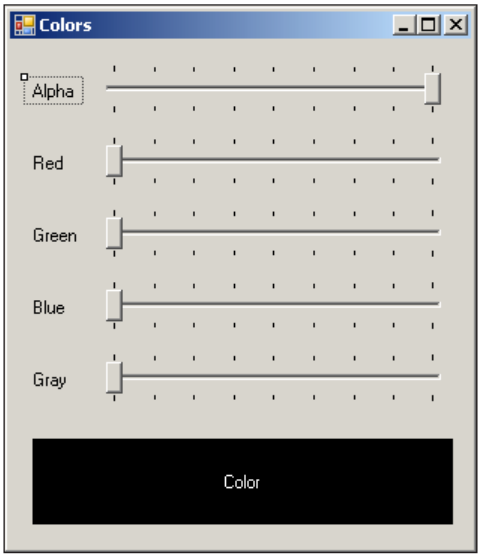


Рис. 9.1. Вид формы `Form1` для проекта COLORS

Листинг 9.1. Настройка свойств

```
Form1: Text = Colors, StartPosition = CenterScreen
trackBar1: LargeChange = 32, Maximum = 255, TickFrequency = 32,
  TickStyle = Both
label6: Text = Color, AutoSize = False, Dock = Fill,
  TextAlign = MiddleCenter, ForeColor = White, BackColor = Black
```

Листинг 9.2. Обработчик `trackBar1.Scroll`

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
  label6.BackColor = Color.FromArgb(trackBar1.Value, trackBar2.Value,
    trackBar3.Value, trackBar4.Value);
}
```

Результат. Цвет фона метки `label6` определяется как комбинация четырех цветовых составляющих, а именно прозрачности (Alpha) и интенсивности трех базовых цветов: красного (Red), зеленого (Green) и синего (Blue). Каждая цветовая составляющая может меняться в пределах от 0 до 255; значение 255 для составляющей Alpha соответствует полной непрозрачности. В нашей про-

грамме значения цветовых составляющих задаются положением соответствующего компонента `TrackBar` (нижний компонент `trackBar5` пока не используется). Поскольку панель с меткой `label6` содержит фоновый рисунок, этот рисунок "просвечивает" сквозь фон метки при уровне прозрачности, отличном от 255.

Комментарии

1. Хотя дизайнер среды Visual C# обеспечивает удобные возможности по выравниванию компонентов друг относительно друга (а также относительно границ формы) в ходе простого перетаскивания компонентов по форме, в некоторых случаях панель выравнивания **Layout** оказывается очень полезной. В частности, с ее помощью легко выровнять выделенный компонент (или группу выделенных компонентов как одно целое) горизонтально или вертикально по центру формы, пропорционально увеличить горизонтальные или вертикальные промежутки между компонентами, установить для выделенных компонентов одинаковую ширину или высоту, изменить *z*-координату компонента, переместив его "вверх" или "вниз" (о *z*-порядке см. комментарий в *разд. 10.1*). Наконец, с помощью крайней правой кнопки  на данной панели можно быстро вызвать режим **Tab Order** настройки порядка обхода компонентов (см. комментарий в *разд. 8.1*).
2. Компонент `TrackBar`, называемый *ползунком*, удобно использовать в ситуациях, когда требуется задать параметр, принимающий значения целого типа из некоторого (не слишком большого) диапазона. При настройке компонента `trackBar1` в нашем проекте мы установили значения следующих его свойств: `LargeChange` (шаг при нажатии клавиш <PgUp> и <PgDn>), `Maximum` (максимальное допустимое значение), `TickFrequency` (частота следования штрихов), `TickStyle` (стиль расположения штрихов), а также `Value` (текущее значение ползунка). Другие свойства изменять не потребовалось, так как нас устроили их значения по умолчанию. Перечислим некоторые из таких свойств: `SmallChange` (шаг ползунка при нажатии клавиш со стрелками, по умолчанию равен **`1`**), `Minimum` (минимальное допустимое значение, равное по умолчанию **`0`**) и `Orientation` (определяет ориентацию ползунка; по умолчанию ориентация горизонтальная: **`Horizontal`**). Заметим, что изменение ползунка на величину `LargeChange` выполняется не только при нажатии клавиш <PgUp> или <PgDn>, но и при щелчке мышью слева или справа от *бегунка* (thumb), отмечающего текущее значение ползунка.
3. Компонент `TrackBar` имеет один недостаток: при изменении его "толщины" (т. е. вертикального размера `Height` в случае горизонтальной ориентации

или горизонтального размера `Width` в случае вертикальной ориентации) не происходит центрирования элементов ползунка относительно его новых границ, а также пропорционального изменения размеров бегунка (заметим, что толщину ползунка можно изменить только при значении свойства `AutoSize`, равном **`False`**). Это делает невозможным использование ползунков небольшого размера.

Если толщина ползунка, установленная по умолчанию, является неподходящей, то можно использовать альтернативные компоненты с аналогичными свойствами: *горизонтальную* (`HScrollBar`) и *вертикальную* (`VScrollBar`) *полосы прокрутки*. При этом, однако, следует иметь в виду два важных обстоятельства.

Во-первых, по умолчанию полоса прокрутки не может принимать фокус (даже при щелчке на ней мышью). Если такое поведение является нежелательным, то следует установить значение ее свойства `TabStop` равным **`True`**.

Во-вторых, максимальное значение, которое может принимать свойство `Value` полосы прокрутки, равно `Maximum – LargeScroll + 1`, т. е. может быть *меньше*, чем `Maximum`. Подобное странное, на первый взгляд, поведение оказывается на самом деле вполне естественным, если принять во внимание тот факт, что значение `LargeScroll` для полосы прокрутки определяет не только шаг при нажатии клавиши <PgUp> или <PgDn>, но и размер (ширину) бегунка относительно всей полосы прокрутки, а значение `Value` соответствует позиции границы бегунка (точнее, левой границы для горизонтальной полосы прокрутки и верхней границы для вертикальной). Поэтому если, к примеру, свойство `Minimum` равно **`1`**, свойство `Maximum` равно **`10`**, а `LargeScroll` равно **`5`**, то бегунок будет иметь ширину, лишь в два раза меньшую ширины области прокрутки, и, значит, при его перемещении до упора вправо (или вниз для вертикальной полосы) его левая (соответственно, верхняя) граница будет располагаться лишь на уровне значения `6` — это значение и будет максимально возможным значением свойства `Value` (см. приведенную выше формулу). Отметим, что подобное поведение хорошо подходит для листания текстовой информации с помощью вертикальной полосы прокрутки, если считать, что значение `Value` соответствует номеру верхней строки текста, отображаемой на экране (так, в приведенном выше примере при `Value = 6` на экране будут отображаться последние 5 строк текста — с 6 по 10 — и дальнейшее листание вниз уже не потребуется).

Если бы полосы прокрутки использовались в нашем примере, то для обеспечения диапазона значений 0—255 для свойства `Value` при сохранении

значения LargeScroll, равного **32**, необходимо было бы установить свойство Maximum равным **286** (т. е. 255 + LargeScroll – 1). Разумеется, все подобные проблемы снимаются, если положить LargeScroll равным **1**, однако в этом случае теряется возможность перебора значений с большим шагом (например, клавишами <PgUp> и <PgDn>).

Завершая краткий обзор особенностей полос прокрутки, отметим, что с помощью обработчика их события Scroll можно очень гибко реагировать на любые действия пользователя, связанные с полосой прокрутки (позволяя, например, не обрабатывать каждое изменение значения Value, если пользователь перетаскивает бегунок с помощью мыши, а отреагировать только на финальное значение Value в момент отпускания бегунка). Событие Scroll компонента TrackBar подобными возможностями не обладает.

9.2. Инвертирование цветов и вывод цветовых констант

Добавьте новые операторы в метод trackBar1_Scroll (листинг 9.3).

Листинг 9.3. Новый вариант метода trackBar1_Scroll

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
  label6.BackColor = Color.FromArgb(trackBar1.Value, trackBar2.Value,
    trackBar3.Value, trackBar4.Value);
  Color c = label6.BackColor;
  label6.ForeColor = Color.FromArgb(0xFF ^ c.R, 0xFF ^ c.G, 0xFF ^ c.B);
  label6.Text = c.Name.ToUpper();
}
```

Результат. Числовое значение текущего цвета в формате ARGB (Alpha, Red, Green, Blue) отображается на панели в виде шестнадцатеричного числа. При этом для каждой цветовой составляющей отводится по два знака, а буквы A—F (соответствующие шестнадцатеричным цифрам от 10 до 15) изображаются в верхнем регистре. Например, значение цвета Maroon (непрозрачный темно-красный цвет интенсивности 128) имеет вид **FF800000**. Цвет текста — непрозрачный и *инверсный* по отношению к цвету фона панели. См. также комментарии 1 и 2.

Недочет 1. При запуске программы метка `label6` содержит текст **Color**, а не числовое значение непрозрачного черного цвета.

Исправление. В конструктор класса `Form1` добавьте оператор:

```
trackBar1_Scroll(null, null);
```

Результат. Теперь метод `trackBar1_Scroll` вызывается в момент создания формы, что обеспечивает правильную настройку внешнего вида метки `label6` (при вызове метода оба его параметра можно положить равными `null`, поскольку в этом методе ни один из параметров не используется).

> ***ПРИМЕЧАНИЕ***
>
> Отмеченный недочет можно было бы исправить, просто положив значение свойства `Text` метки `label6` равным **`FF000000`** (числовое значение непрозрачного черного цвета). Однако использованный вариант исправления является более гибким, поскольку позволяет правильно отображать начальный вид метки при любых изменениях метода `trackBar1_Scroll`, которые могут быть сделаны впоследствии *(см. разд. 9.4)*.

Недочет 2. Если прозрачность имеет значение, меньшее `16`, то в метке `label6` отображается менее 8 цифр (в частности, при полностью прозрачном черном цвете метка будет содержать единственную цифру 0). Подобный способ представления цвета является не вполне естественным; логичнее *всегда* отображать шестнадцатеричное число с 8 знаками (по два знака на каждую цветовую составляющую).

Исправление. В методе `trackBar1_Scroll` замените последний оператор на следующий:

```
label6.Text = c.ToArgb().ToString("X8");
```

Результат. Теперь число `c.ToArgb()` выводится в шестнадцатеричном формате `X` с заглавными латинскими буквами и обязательными 8 знаками (мы использовали форматирующий вариант метода `ToString` для типа `int`).

Комментарии

1. Свойства `A`, `R`, `G`, `B` структуры `Color` (доступные только для чтения) позволяют получить числовое значение соответствующей цветовой составляющей. Для инвертирования каждого из базовых цветов использована побитовая операция `^` (исключающее ИЛИ). При использовании варианта метода `FromArgb` с тремя параметрами `(R, G, B)` предполагается, что прозрачность `A` равна `255`.

2. Метод `ToUpper` класса `string` преобразует все буквенные символы строки к верхнему регистру и возвращает измененную строку (метод, преобразующий буквенные символы к нижнему регистру, имеет имя `ToLower`). Символы, не являющиеся буквами, не изменяются. Заметим, что методы `ToUpper` и `ToLower` правильно обрабатывают не только латинские, но и русские буквы.

9.3. Отображение оттенков серого цвета

Определите обработчик события `Scroll` для компонента `trackBar5` (листинг 9.4).

Листинг 9.4. Обработчик `trackBar5.Scroll`

```
private void trackBar5_Scroll(object sender, EventArgs e)
{
  trackBar2.Value = trackBar3.Value = trackBar4.Value = trackBar5.Value;
}
```

Результат. Перемещение ползунка `trackBar5` обеспечивает синхронное изменение всех трех базовых цветов, давая в итоге различные оттенки серого цвета (значение прозрачности при этом не изменяется).

Ошибка. Несмотря на то, что при перемещении ползунка `trackBar5` положение ползунков `trackBar2`—`trackBar4` синхронно изменяется, это изменение не влияет на цвет метки `label6`. Отмеченная ошибка объясняется тем обстоятельством, что при *программном* изменении свойства `Value` событие `Scroll` не возникает.

Исправление. Добавьте в метод `trackBar5_Scroll` оператор:

```
trackBar1_Scroll(null, null);
```

> ***ПРИМЕЧАНИЕ***
>
> Если программа должна не только реагировать на изменения положения ползунков, сделанные пользователем, но и сама изменять ползунки, то целесообразнее использовать обработчик события `ValueChanged`. Так, если бы в нашем проекте мы определили для компонентов `trackBar1`—`trackBar4` обработчик события `ValueChanged` вместо обработчика события `Scroll`, то уже первый вариант метода `trackBar5_Scroll` (см. листинг 9.4) работал бы правильно.

9.4. Вывод цветовых имен

Измените метод `trackBar1_Scroll` (листинг 9.5).

Листинг 9.5. Новый вариант метода `trackBar1_Scroll`

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
  label6.BackColor = Color.FromArgb(trackBar1.Value, trackBar2.Value,
    trackBar3.Value, trackBar4.Value);
  Color c = label6.BackColor;
  label6.ForeColor = Color.FromArgb(0xFF ^ c.R, 0xFF ^ c.G, 0xFF ^ c.B);
  string s = c.ToArgb().ToString("X8");
  switch (c.A)
  {
    case 0:
      s += " Transparent";
      break;
    case 255:
      string
        s0 = ColorTranslator.FromWin32(ColorTranslator.ToWin32(c)).Name;
      if (s0.Substring(0, 2) != "ff")
        s += " " + s0;
      break;
  }
  label6.Text = s;
}
```

Результат. В том случае, когда с текущим цветом связано определенное имя (например, **Black** или **Maroon**), на панели отображается не только числовое значение текущего цвета в шестнадцатеричном формате, но и его имя. Если прозрачность имеет значение, равное 0, то рядом с числовым значением цвета выводится текст **Transparent**.

Комментарии

1. Все цвета, имеющие имена *(именованные цвета)*, содержатся в перечислении `KnownColor`. Структура `Color` имеет метод `ToKnownColor`, возвращающий для любого именованного цвета соответствующий ему элемент перечисления `KnownColor` или `0`, если цвет не является именованным. Однако область применения метода `ToKnownColor` существенно ограничивается

тем обстоятельством, что если цвет создан с помощью метода FromArgb, то вызов для него метода ToKnownColor обязательно вернет 0 (даже если цвет с указанными составляющими входит в набор именованных цветов). Подобное поведение является одним из проявлений важной особенности структуры Color: при анализе (в частности, сравнении) структур типа Color учитываются не только их цветовые составляющие, но и *способ*, с помощью которого эти структуры были созданы. Например, если структура c1 типа Color была создана с помощью метода FromArgb(0, 0, 0), а структура c2 — с помощью метода FromName("Black"), то они будут считаться различными, хотя обе представляют непрозрачный черный цвет (заметим также, что выражение c1.Name вернет строку ff000000, а выражение c2.Name — строку Black).

2. Поскольку непосредственное применение метода ToKnownColor в нашем случае не позволит получить требуемый результат, можно было бы сформировать массив всех именованных цветов (воспользовавшись для этого перечислением KnownColor — *см. разд. 25.1*), а затем в цикле сравнивать цветовые характеристики каждого именованного цвета с цветовыми характеристиками интересующего нас цвета. Однако в .NET имеется класс, позволяющий распознать именованный цвет более простым способом: это ColorTranslator. Если создать цвет с помощью метода FromWin32 класса ColorTranslator, то в случае именованного цвета его свойство Name вернет имя цвета, а в противном случае Name вернет числовое представление цвета в формате ARGB. Для возможности применения к объекту типа Color метода FromWin32 этот объект необходимо предварительно преобразовать к целочисленному формату RGB с помощью метода ToWin32. Правда, при использовании методов класса ColorTranslator не будет учитываться прозрачность, так как в цветовом формате для Win32 (RGB) подобная составляющая цвета не предусмотрена. Поэтому мы пользуемся методами класса ColorTranslator только для непрозрачных цветов, т. е. цветов, для которых Alpha = 255. Заметим, что такой подход является вполне допустимым, поскольку все именованные цвета, за исключением полностью прозрачного цвета с именем Transparent, являются непрозрачными. Кроме того, мы особым образом обрабатываем ситуацию полной прозрачности Alpha = 0.

3. Проверка s0.Substring(0, 2) != "ff" позволяет определить, что содержит строка s0: имя цвета или его числовое представление. В последнем случае строка s0 всегда начинается с двух символов ff, соответствующих значению Alpha = 255 (полная непрозрачность). Для получения подстроки строки s0 используется метод Substring(start, len) класса string, возвращающий подстроку длины len, начинающуюся с символа с индексом start. Имеется вариант данного метода с одним параметром start; этот вариант возвращает всю оставшуюся часть строки, начиная с символа с индексом start.

9.5. Связывание компонентов с подписями к ним

Измените свойства `Text` для меток `label1`—`label5`, добавив в них символ `&`: **`&Alpha`**, **`&Red`**, **`&Green`**, **`&Blue`**, **`Gra&y`** (для метки `Gray` символ `&` выделяет последнюю букву, поскольку все предыдущие буквы уже использованы в качестве выделенных символов других меток).

Используя команду меню **View | Tab Order** (см. комментарий 1 в *разд. 8.1*), установите значения свойства `TabIndex` для следующих компонентов, размещенных в форме: `label1` — **`0`**, `trackBar1` — **`1`**, `label2` — **`2`**, `trackBar2` — **`3`**, `label3` — **`4`**, `trackBar3` — **`5`**, `label4` — **`6`**, `trackBar4` — **`7`**, `label5` — **`8`**, `trackBar5` — **`9`**.

Результат. Переключение между ползунками теперь можно осуществлять с помощью клавиатурных <Alt>-комбинаций символов, подчеркнутых в подписях к ползункам (<Alt>+<A> для ползунка, определяющего прозрачность, <Alt>+<R> для ползунка, определяющего интенсивность красного цвета, и т. д.). Отметим, что <Alt>-комбинации выполняют требуемое действие только при установке английской раскладки клавиатуры.

> ***ПРИМЕЧАНИЕ***
>
> Если при запуске программы в метках отсутствуют линии подчеркивания, то следует нажать клавишу <Alt>. Кроме того, у текущего ползунка может отсутствовать рамка выделения. Для отображения в окне рамки выделения надо нажать клавишу <Tab>.

Комментарий

Если <Alt>-комбинация назначена компоненту, который не может принимать фокус (именно такими компонентами являются метки), то при нажатии подобной комбинации делается попытка установить фокус на компоненте, имеющем большее значение свойства `TabIndex`. Если несколько компонентов имеют одинаковые значения свойства `TabIndex` (что допускается), то они перебираются в *z*-порядке, т. е. по возрастанию *z*-координаты (по поводу *z*-порядка см. комментарий 1 в *разд. 10.1*).

9.6. Привязка компонентов

Измените свойство `Anchor` для компонентов `trackBar1`—`trackBar5`, положив его равным **`Top, Left, Right`** (при установке свойства `Anchor` в окне **Properties** отображается специальная панель; для настройки в этой панели требуемого

варианта следует выделить линию, ведущую к *правой* границе панели, поскольку линии, ведущие к верхней и левой границе, уже выделены по умолчанию).

Аналогичными действиями установите свойство `Anchor` компонента `panel1` равным **`Top, Bottom, Left, Right`** (в данном случае в панели настройки свойства `Anchor` надо выделить линии, ведущие к нижней и правой границам, оставив также выделенными линии, ведущие к верхней и левой границам). Подчеркнем, что необходимо настроить свойство `Anchor` *панели* `panel1`, а не содержащейся на ней метки `label6`.

Результат. Если при выполнении программы изменить размер формы `Form1`, то размер компонентов будет скорректирован в соответствии с новым размером формы (ползунки `trackBar1`—`trackBar5` изменят ширину, а панель `panel1` с меткой `label6` — ширину и высоту). См. также комментарий 1.

Недочет. При уменьшении размера формы может возникнуть ситуация, когда цветовая метка перестанет быть видна, а ползунки станут слишком узкими.

Исправление. Добавьте в конструктор класса `Form1` оператор

```
MinimumSize = Size;
```

Результат. Теперь при выполнении программы размер формы можно только увеличить (по сравнению с исходным), поскольку минимально допустимый размер формы, определяемый свойством `MinimumSize`, совпадает с исходным размером формы, хранящимся в свойстве `Size` (см. комментарий 2).

Комментарии

1. За *привязку* (anchoring) компонентов отвечает свойство `Anchor`, имеющееся у всех визуальных компонентов. С его помощью можно указать, к каким границам формы (а точнее, родительского компонента-контейнера) следует установить привязку. По умолчанию большинство компонентов привязывается к верхней и левой границе своих родительских компонентов. При изменении размеров родительского компонента расстояние от его "привязанных" границ до границ дочернего компонента остается неизменным.

   Если компонент не привязан ни к одной из вертикальных границ, то при изменении размера родительского компонента *ширина* дочернего компонента меняться не будет, а его *расстояние* до левой и правой границы родительского компонента будет изменяться синхронно с шириной родительского компонента. Аналогичный эффект достигается и при отсутствии привязки к горизонтальным границам.

   Приведем пример. Если разместить компонент по центру формы и установить привязку ко всем границам, то при любом изменении размеров формы

он будет оставаться в центре формы, причем его размеры будут изменяться. Если же для такого компонента полностью снять привязку (т. е. установить для его свойства `Anchor` значение **`None`**), то компонент по-прежнему будет оставаться в центре формы, но его размер при этом меняться не будет.

Другим способом обеспечить синхронизацию размера и положения компонентов относительно родительского компонента (в частности, формы) является *стыковка* (docking), настраиваемая с помощью свойства `Dock` (стыковка подробно рассматривается в *разд. 13.3*). Эти способы являются взаимно исключающими: если изменить значение по умолчанию для свойства `Anchor` или `Dock`, то другое свойство автоматически примет значение по умолчанию.

2. Помимо свойства `MinimumSize`, использованного при исправлении недочета, имеется аналогичное свойство `MaximumSize`, которое позволяет ограничить максимальный размер формы. Если по какому-либо измерению размер ограничивать не требуется, то достаточно присвоить соответствующей координате свойств `MinimumSize` и `MaximumSize` значение **`0`**. Заметим, что данные свойства имеются у всех визуальных компонентов; по умолчанию каждое из них равно **`0; 0`**, т. е. ограничение на размер отсутствует.

Глава 10

Обработка событий от мыши: проект MOUSE

Проект MOUSE знакомит с обработкой событий, связанных с манипулятором "мышь". Рассматривается явление захвата мыши компонентом и проблемы, порождаемые подобным захватом. Кроме того, обсуждаются вопросы, связанные с отношением "родитель" — "дочерний компонент" и с порядком расположения дочерних компонентов в форме или другом компоненте-контейнере *(z-порядок)*. Также описываются возможности среды Visual C#, связанные с поиском и заменой.

10.1. Перетаскивание с помощью мыши. Настройка *z*-порядка компонентов

После создания проекта MOUSE разместите в форме `Form1` два компонента типа `Panel` (они получат имена `panel1` и `panel2`) и настройте свойства формы и добавленных компонентов (листинг 10.1, рис. 10.1).

В начало описания класса `Form1` добавьте описание поля:

```
private Point p;
```

Определите обработчики событий `MouseDown` и `MouseMove` для панели `panel1` (листинг 10.2), после чего свяжите созданные обработчики с событиями `MouseDown` и `MouseMove` панели `panel2` (о связывании обработчиков с несколькими компонентами *см. в разд. 6.1*). Кроме того, определите обработчик события `MouseMove` для формы `Form1` (листинг 10.3).

Листинг 10.1. Настройка свойств

```
Form1: Text = Mouse, MaximizeBox = False, FormBorderStyle = FixedSingle,
  StartPosition = CenterScreen
panel1: BackColor = Red, BorderStyle = Fixed3D
panel2: BackColor = Green, BorderStyle = Fixed3D
```

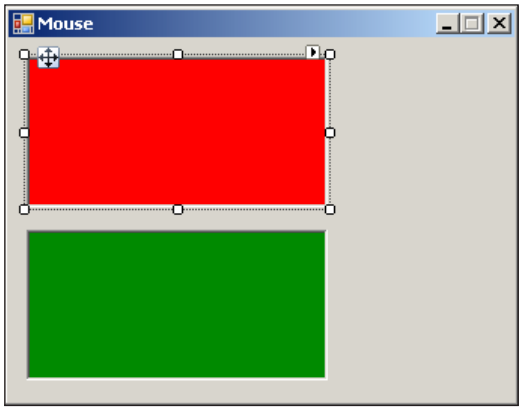


Рис. 10.1. Вид формы `Form1` для проекта MOUSE на начальном этапе разработки

Листинг 10.2. Обработчики `panel1.MouseDown` и `panel1.MouseMove`

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
}
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
  if (e.Button == MouseButtons.Left)
    a.Location += s0;
}
```

Листинг 10.3. Обработчик `Form1.MouseMove`

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
  Text = "Mouse";
}
```

Результат. После запуска приложения перемещение курсора мыши над любой панелью приводит к тому, что в заголовке окна, кроме текста **Mouse**,

выводится имя панели и текущие значения *локальных* координат мыши относительно панели. Если при этом удерживается нажатой левая кнопка мыши, то панель перемещается по форме ("перетаскивается" мышью). См. также комментарии 1—3.

Недочет. Панель `panel1` при перетаскивании может заслоняться панелью `panel2`.

Исправление. Добавьте в метод `panel1_MouseDown` два оператора:

```
Panel a = sender as Panel;
a.BringToFront();
```

Результат. Теперь перетаскиваемая панель всегда располагается поверх остальных компонентов окна (см. комментарий 4).

> ***ПРИМЕЧАНИЕ***
>
> Требуемый результат можно получить с помощью единственного оператора `(sender as Panel).BringToFront()`. Мы разделили этот оператор на два, описав вспомогательный объект `a` типа `Panel`, поскольку в дальнейшем метод `panel1_MouseDown` будет дополнен другими операторами, использующими объект `a`.

Комментарии

1. Для формирования заголовка формы при перетаскивании панели (листинг 10.2, метод `panel1_MouseMove`) используется метод `Format` класса `string`, возвращающий строку, которая содержит как фиксированный текст, так и строковые представления различных объектов, отформатированные требуемым образом. Первым параметром метода `Format` является форматная строка, содержащая обычный текст и *форматные настройки* для остальных параметров (количество форматируемых параметров может быть произвольным). Форматные настройки заключаются в фигурные скобки `{}`. В нашем случае использованы простейшие форматные настройки, в которых задается только порядковый номер параметра, выводимого в указанной позиции форматной строки (в подобной простейшей ситуации для форматирования параметра автоматически вызывается его метод `ToString`). Обратите внимание на то, что метод `ToString` объекта типа `Point` снабжает координаты `X` и `Y` комментариями вида `X=` и `Y=` и заключает полученный текст в фигурные скобки, например, `{X=50,Y=15}`. По поводу форматных настроек см. также комментарий 4 в *разд. 2.2*.
2. В методе `panel1_MouseMove` (см. листинг 10.2) при вычислении нового положения панели (свойство `Location` типа `Point`) к его прежнему положению

прибавляется смещение `s0` типа `Size`, которое создается с помощью конструктора структуры `Size`. Объекты типа `Size` можно не только прибавлять, но и вычитать из объекта типа `Point`; результатом будет новый объект типа `Point`. Складывать и вычитать объекты типа `Point` нельзя. Объекты типа `Size` можно складывать и вычитать; результатом будет объект типа `Size`. Заметим, что типы `Point` и `Size` можно преобразовывать один в другой с помощью операции явного приведения типа. Учитывая этот факт, можно определить новое положение панели другим способом:

```
a.Location += (Size)e.Location - (Size)p;
```

3. Кажущееся постоянство значений координат, отображаемых на экране при перетаскивании панели, объясняется тем, что немедленно после изменения координат происходит корректировка положения панели, а вместе с тем и пересчет локальных координат мыши относительно панели. Если перемещать панель достаточно быстро, то можно заметить, что в заголовке формы на короткое время отобразятся другие значения координат.

4. Любой визуальный компонент имеет метод `BringToFront`, "поднимающий" его над всеми компонентами, и метод `SendToBack`, "опускающий" его под все компоненты. Эти действия связаны с изменением *z-порядка* компонентов, т. е. их относительного расположения на оси z, ориентированной перпендикулярно плоскости экрана. Понятие z-порядка тесно связано с отношением *"родитель — дочерний компонент"* и имеет смысл только для дочерних компонентов одного и того же родителя, поэтому вначале следует обсудить вопросы, связанные с отношением *"родитель — дочерний компонент"*.

   Как правило, родителем компонентов является форма, однако в качестве родителей могут выступать любые компоненты-*контейнеры* (подобные компоненты указаны в группе **Containers** окна **Toolbox**). Далее в этом комментарии будем использовать переменную `p` (от англ. *parent* — родитель) для обозначения родителя, т. е. формы или компонента-контейнера.

   Все дочерние компоненты содержатся в свойстве-коллекции `Controls` своего родителя `p`. Свойство `Controls` имеет тип `ControlCollection` и позволяет обратиться к любому дочернему компоненту либо по целочисленному индексу, начиная с нулевого (например, `p.Controls[0]`), либо по строковому ключу, совпадающему с именем дочернего компонента (т. е. со значением его свойства `Name`, например, `p.Controls["button1"]`). Для добавления компонента в коллекцию `Controls` можно использовать ее метод `Add` (например, `p.Controls.Add(button1)`). Можно также положить свойство `Parent` компонента равным `p` (например, `button1.Parent = p`). Эти способы добавления компонента в коллекцию `Controls` полностью эквивалентны. Новый компонент добавляется в *конец* коллекции `Controls` (начальным

элементом коллекции считается элемент с индексом `0`). Размер коллекции `Controls` можно определить с помощью ее свойства `Count`, доступного только для чтения.

Положение дочернего компонента определяется относительно левого верхнего угла клиентской части родителя (т. е. задается в локальной системе координат родителя). Дочерние компоненты не могут быть отображены вне видимой области своего родителя (хотя они могут размещаться вне видимой области — *см. разд. 10.2*).

Следует заметить, что форма обычно не имеет родителя; в этом случае ее свойство `Parent` равно `null`, а ее положение задается в экранных (т. е. глобальных) координатах. Исключение составляют *дочерние формы* в MDI-приложении *(см. главу 27)*. Кроме того, форма может иметь *владельца* (owner) — *см. разд. 5.1*, *5.4*, *5.6*.

Вернемся к понятию *z*-порядка. Удобно считать, что ось *z* направлена от пользователя, т. е. внутрь экрана (при этом мы получим *правую* систему координат *xyz*, поскольку ось *y* направлена на экране вертикально вниз). При таком предположении можно считать индексы дочерних компонентов в коллекции `Controls` их *z*-координатами. Иными словами, первый элемент коллекции `Controls` (имеющий индекс `0`) располагается на оси *z* в точке 0, второй компонент (с индексом `1`) — в точке 1 и т. д. Подчеркнем, что ось *z* направлена от пользователя, поэтому на экране первый дочерний компонент будет выглядеть самым верхним и может заслонять второй и прочие дочерние компоненты.

Это утверждение, казалось бы, противоречит тому результату, который мы получаем, добавляя компонент в форму в режиме дизайна (ведь именно компонент, размещаемый в форме в последнюю очередь, оказывается самым верхним). Дело в том, что добавление компонента в форму в режиме дизайна и добавление компонента к коллекции `Controls` *приводят к разным результатам*. В этом можно убедиться, открыв файл Form1.Designer.cs и просмотрев текст метода `InitializeComponent` (предварительно может потребоваться развернуть фрагмент файла Form1.Designer.cs, помеченный как **Windows Form Designer generated code**). Оказывается, методы `Controls.Add` вызываются для компонентов в порядке, *обратном* их помещению в форму. Иными словами, при размещении компонента в форме в режиме дизайна данный компонент располагается не в конце, а в начале коллекции `Controls`. Это обстоятельство объясняет кажущееся противоречие. Кроме того, четкое понимание этого обстоятельства позволяет избежать ряда ошибок, связанных с размещением компонентов в форме и доступом к ним с помощью коллекции `Controls`. Укажем две наиболее характерные ошибки подобного рода.

Не следует считать, что первый компонент, помещенный в форму в режиме дизайна, будет иметь в коллекции `Controls` индекс `0`. Наоборот, индекс `0` на каждом этапе разработки программы будет иметь *последний* компонент, добавленный в форму в режиме дизайна. Именно поэтому к элементам коллекции `Controls` удобнее обращаться не по индексам, а по строкам-ключам вида `"button1"`.

Не следует считать, что компонент, добавленный в форму не в режиме дизайна, а программным путем (т. е. посредством явного присваивания объекта-формы его свойству `Parent` или вызова метода `Add` коллекции `Controls` формы), будет выглядеть как самый верхний компонент. Наоборот, он будет помещен в конец коллекции `Controls` и, следовательно, будет находиться *под* всеми остальными дочерними компонентами, т. е. будет последним в z-порядке. Для того чтобы компонент стал самым верхним, необходимо после его добавления в коллекцию `Controls` вызвать для него метод `BringToFront`.

Четкое понимание особенностей z-порядка, принятого в .NET, позволяет также разобраться с тонкостями *стыковки* компонентов (т. е. их привязки к границам формы, выполняемой с помощью настройки свойства `Dock`). Однако рассмотрение этого вопроса мы отложим до другого примера *(см. разд. 13.3)*.

10.2. Изменение размеров с помощью мыши

В начало описания класса `Form1` добавьте описание нового поля:

```
private Size s;
```

Добавьте новые операторы в методы `panel1_MouseDown` и `panel1_MouseMove` (листинг 10.4).

Листинг 10.4. Новые варианты методов `panel1_MouseDown` и `panel1_MouseMove`

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
  Panel a = sender as Panel;
  a.BringToFront();
  s = a.Size;
```

```
}
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
  if (e.Button == MouseButtons.Left)
    a.Location += s0;
  else
    if (e.Button == MouseButtons.Right)
      a.Size = s + s0;
}
```

Результат. Если перемещать курсор мыши над панелью, удерживая нажатой правую кнопку мыши, то происходит изменение размеров панели (левая кнопка по-прежнему используется для изменения положения панели). Следует обратить внимание на то, что перетащить панель можно даже на область вне окна. Если при этом не отпускать кнопку мыши, то панель можно вернуть обратно на видимую часть окна (аналогично, уменьшив размеры панели до нулевых, можно затем восстановить их). Отмеченная особенность связана с явлением *захвата мыши* (mouse capture): если нажать над компонентом какую-либо кнопку мыши, то этот компонент захватит мышь и "заставит" ее передавать ему все сообщения (даже если курсор мыши покинет компонент) до тех пор, пока кнопка мыши не будет отпущена (впрочем, это событие тоже будет обработано компонентом, ранее захватившим мышь). См. также комментарий 1.

Недочет. Если панель перетащить целиком за границы окна и *отпустить* кнопку мыши, то доступ к панели окажется невозможным. Недоступной панель станет и при уменьшении ее размеров до нулевых, если при этом отпустить кнопку мыши.

Исправление. Установите свойство `AutoScroll` формы `Form1` равным **`True`** и измените метод `panel1_MouseMove` (листинг 10.5).

Листинг 10.5. Исправленный вариант метода `panel1_MouseMove`

```
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
```

```
  if (e.Button == MouseButtons.Left)
  {
    Point p0 = a.Location + s0;
    a.Location = new Point(Math.Max(0, p0.X), Math.Max(0, p0.Y));
  }
  else
    if (e.Button == MouseButtons.Right)
    {
      s0 += s;
      a.Size = new Size(Math.Max(50, s0.Width), Math.Max(20, s0.Height));
    }
}
```

Результат. Теперь перетаскивать панель за пределы левой или верхней границы окна невозможно, и, кроме того, определен минимальный размер панели (в пикселах), равный 50 × 20. Перетаскивание панели за пределы правой или нижней границы окна также не делает ее недоступной, поскольку при этом на окне автоматически появляются полосы прокрутки (благодаря значению **`True`** для свойства формы `AutoScroll`). См. также комментарии 2 и 3.

Комментарии

1. Эффект захвата мыши можно продемонстрировать в нашей программе еще одним способом: если нажать кнопку мыши на свободной части формы, а затем переместить курсор мыши на одну из панелей, то заголовок формы не изменится (он по-прежнему будет иметь вид **Mouse**). Это происходит потому, что в данной ситуации мышь захвачена самой формой, и даже при перемещении мыши над панелью событие `MouseMove` передается не этой панели, а захватившей мышь форме. Стоит в этой ситуации отпустить мышь, находясь на панели, как любое ее перемещение уже будет перехвачено панелью, что проявится в изменении заголовка формы.
2. При исправлении недочета мы ограничили размеры компонента, добавив соответствующий код в программу (см. листинг 10.5). Ограничить размеры компонента (а также формы — *см. разд. 9.6*) можно также с помощью его свойств `MinimumSize` и `MaximumSize` типа `Size`. В нашем случае достаточно положить свойство `MinumumSize` панелей `panel1` и `panel2` равным **`50; 20`**, используя окно **Properties**. Однако такой способ требует настройки свойства `MinumumSize` для *каждого* компонента, размеры которого можно изменять. Кроме того, свойств, ограничивающих *положение* компонента, не предусмотрено.

3. Функция Max, использованная в листинге 10.5, возвращает максимальный из двух своих параметров числового типа. Эта функция является статическим методом класса Math. Все прочие стандартные математические функции также реализованы в виде статических методов класса Math.

10.3. Использование дополнительных курсоров

Добавьте новые операторы в метод panel1_MouseDown (листинг 10.6). Определите обработчик события MouseUp для панели panel1 (листинг 10.7), после чего свяжите созданный обработчик с событием MouseUp панели panel2.

Листинг 10.6. Новый вариант метода panel1_MouseDown

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
  Panel a = sender as Panel;
  a.BringToFront();
  s = a.Size;
  if (e.Button == MouseButtons.Left)
    a.Cursor = Cursors.Hand;
  else
    if (e.Button == MouseButtons.Right)
      a.Cursor = Cursors.SizeNWSE;
}
```

Листинг 10.7. Обработчик panel1.MouseUp

```
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  (sender as Panel).Cursor = null;
}
```

Результат. В режиме изменения размеров панели курсор мыши принимает вид диагональной двунаправленной стрелки, а в режиме перетаскивания — "указывающей руки". После выхода из этих режимов восстанавливается стандартный вид курсора.

ПРИМЕЧАНИЕ

Свойство `Cursor`, имеющееся у любого визуального компонента, а также класс `Cursors` были подробно рассмотрены в *главе 7*. Напомним, что если свойство `Cursor` равно `null`, то компонент будет использовать курсор своего родителя (в данном случае формы).

10.4. Обработка ситуации с одновременным нажатием двух кнопок мыши: первый вариант

Наша программа будет прекрасно работать до тех пор, пока пользователю не придет в голову мысль нажать левую кнопку мыши при нажатой правой кнопке (или наоборот). Для определенности опишем поведение программы в ситуации, когда при нажатой на панели левой кнопке мыши была дополнительно нажата правая (и, кроме того, будем пока считать, что при выполнении описываемых далее действий курсор мыши не будет выходить за пределы панели, на которой он первоначально находился). В момент нажатия второй кнопки вид курсора изменится на диагональную стрелку, однако при последующем перемещении мыши ни положение, ни размер панели изменяться не будут. Если отпустить одну из кнопок мыши, то курсор примет вид стандартной стрелки (вид курсора по умолчанию), однако при перемещении мыши будет выполняться действие, определяемое той кнопкой мыши, которая осталась нажатой.

Подобное поведение объясняется следующими особенностями обработчиков событий от мыши: при выполнении обработчиков для событий `MouseDown` и `MouseUp` параметр `e.Button` содержит информацию о той кнопке мыши, которая только что была нажата (или, соответственно, отпущена). Информация о прочих нажатых кнопках в параметре `e.Button` не содержится. С другой стороны, при выполнении обработчика события `MouseMove` параметр `e.Button` содержит информацию обо *всех* нажатых в данный момент кнопках мыши; при этом элементы перечисления `MouseButtons`, соответствующие нажатым кнопкам, объединяются операцией `|` (побитовое ИЛИ).

Последнее обстоятельство позволяет понять, почему при одновременно нажатых кнопках наша программа не выполняет никаких действий: в самом деле, действия по перемещению или изменению размера панели реализуются в обработчике события `MouseMove` при выполнении соответственно условий `e.Button == MouseButtons.Left` или `e.Button == MouseButtons.Right`, но если нажаты обе кнопки мыши, то ни одно из этих условий не является истинным, поскольку в этом случае свойство `e.Button` содержит выражение `MouseButtons.Left | MouseButtons.Right`.

Внесем в программу изменения, которые позволят более естественным образом обрабатывать ситуацию, связанную с одновременным нажатием левой и правой кнопок мыши (листинг 10.8).

Листинг 10.8. Новые варианты методов `panel1_MouseDown` и `panel1_MouseUp`

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
  Panel a = sender as Panel;
  a.BringToFront();
  s = a.Size;
  if (Control.MouseButtons != MouseButtons.Left
    && Control.MouseButtons != MouseButtons.Right)
    a.Cursor = null;
  else
    if (e.Button == MouseButtons.Left)
      a.Cursor = Cursors.Hand;
    else
      if (e.Button == MouseButtons.Right)
        a.Cursor = Cursors.SizeNWSE;
}
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  if (Control.MouseButtons == MouseButtons.Left)
    a.Cursor = Cursors.Hand;
  else
  if (Control.MouseButtons == MouseButtons.Right)
    a.Cursor = Cursors.SizeNWSE;
  else
    a.Cursor = null;
}
```

Результат. Как и ранее, при одновременно нажатых левой и правой кнопках мыши никакое действие не выполняется. Однако теперь это согласуется с тем, что в данной ситуации курсор мыши имеет вид курсора по умолчанию. Далее, при отпускании одной из двух нажатых кнопок курсор принимает вид, соответ-

ствующий действию для той кнопки мыши, которая остается нажатой, и при перемещении мыши это действие выполняется.

Комментарий

Чтобы правильно определить вид курсора при нажатии или отпускании кнопки мыши, недостаточно знать, какая кнопка была нажата или отпущена; необходимо также определить, какие кнопки остаются нажатыми. Для этого в программе используется статическое свойство `MouseButtons` класса `Control`, которое в любой момент позволяет определить, какие кнопки мыши являются нажатыми. Заметим, что по своему назначению свойство `MouseButtons` родственно свойству `ModifierKeys` *(см. разд. 4.2)*.

10.5. Обработка ситуации с одновременным нажатием двух кнопок мыши: второй вариант

Исправления, сделанные в *разд. 10.4*, не решают всех проблем, связанных с одновременным нажатием двух кнопок мыши. Так, если при двух нажатых кнопках переместить курсор мыши в другую позицию панели, а затем отпустить одну из кнопок, то, в зависимости от того, какая кнопка осталась нажатой, произойдет скачкообразное изменение либо положения, либо размеров данной панели.

Для исправления этого недочета в очередной раз изменим метод `panel1_MouseUp` (листинг 10.9).

Листинг 10.9. Новый вариант метода `panel1_MouseUp`

```
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  if (Control.MouseButtons == MouseButtons.Left)
    panel1_MouseDown(sender, new MouseEventArgs(MouseButtons.Left,
      0, e.X, e.Y, 0));
  else
  if (Control.MouseButtons == MouseButtons.Right)
    panel1_MouseDown(sender, new MouseEventArgs(MouseButtons.Right,
```

```
      0, e.X, e.Y, 0));
  else
    a.Cursor = null;
}
```

Результат. Теперь в ситуации, описанной выше, скачкообразного изменения свойств панели не происходит.

Комментарий

В момент нажатия кнопки мыши наша программа запоминает информацию, связанную с текущим положением курсора мыши. В дальнейшем эта информация используется при выполнении действий по перемещению или изменению размеров захваченной мышью панели. Отпускание одной из двух нажатых кнопок также требует сохранения информации, связанной с текущим положением курсора мыши, поскольку эта информация будет использована при выполнении действий, за которые отвечает кнопка, остающаяся нажатой. Для сохранения информации в подобной ситуации проще всего вызвать уже имеющийся обработчик для события `MouseDown`, передав ему необходимые параметры. Обратите внимание на то, что второй параметр (типа `MouseEventArgs`) формируется с помощью вызова конструктора, которому передается вся необходимая информация, а именно: какая кнопка остается нажатой и в какой позиции находится курсор мыши. Два оставшихся целочисленных свойства второго параметра в обработчике `panel1_MouseDown` не используются, поэтому для них в конструкторе указываются нулевые значения. Заметим, что первое из этих свойств (`Clicks`, второй параметр конструктора) используется в событиях `MouseDown` и `MouseUp` и позволяет определить, связано ли соответствующее событие с двойным щелчком мыши *(см. разд. 12.3)*, а оставшееся свойство (`Delta`, пятый параметр конструктора) используется только в обработчиках события `MouseWheel`, связанного с вращением колесика мыши.

10.6. Обработка ситуации с одновременным нажатием двух кнопок мыши: третий вариант

К сожалению, проблемы, связанные с одновременным нажатием левой и правой кнопок мыши, на этом не заканчиваются. Предположим теперь, что пользователь выполнит такую последовательность действий:

1. Вначале пользователь нажмет обе кнопки мыши над одной из панелей.

2. Затем он переместит курсор мыши на свободную часть формы (это можно сделать, так как при двух нажатых кнопках перемещение мыши не выполняет действий, связанных с изменением положения или размеров панели).
3. На этой свободной части формы пользователь отпустит одну из кнопок (для определенности, будем считать, что отпущена правая кнопка).
4. Затем, не отпуская кнопки мыши, оставшейся нажатой, пользователь опять переместит курсор мыши на панель.

После выполнения действия 3 перемещение курсора мыши по свободной части формы никак не будет влиять на положение панели (курсор будет иметь стандартный вид). Однако после выполнения действия 4 панель "прыгнет" на новое место, причем величина и направление прыжка будет зависеть от того, в каком месте формы пользователь *отпустил правую кнопку мыши*. Если в результате прыжка панель будет задвинута в левый верхний угол формы и удастся навести на нее курсор мыши, то мы увидим, что он имеет вид руки. Однако стоит его вернуть обратно на форму, как он опять примет стандартный вид.

Все эти странности объясняются тем, что режим захвата мыши завершается, как только вне компонента, захватившего мышь, будет отпущена *какая-либо* кнопка мыши. При отпускании левой кнопки мыши над формой мышь освободится от захвата панелью, однако не будет захвачена формой (поскольку для захвата надо *нажать* на кнопку). Заметим, что перед освобождением от захвата будет вызван обработчик события `MouseUp` для той панели, которая ранее захватила мышь, и в результате вызова этого обработчика (листинг 10.9) будет вызван метод `panel1_MouseDown` для этой панели. Таким образом, для данной панели будет сохранена информация, привязанная к той точке, в которой была отпущена кнопка мыши. Так как теперь мышь не является захваченной, ее перемещение по свободной части формы обрабатывается самой формой (при этом ничего не происходит, за исключением того, что заголовок формы постоянно имеет вид **Mouse**). Но если подобная "незахваченная" мышь окажется над панелью (причем над *любой* панелью), то сработает обработчик `MouseMove` этой панели, а поскольку в данный момент левая кнопка мыши все еще является нажатой, эта панель будет перемещена на новое место с учетом той информации, которая была сохранена в момент отпускания правой кнопки мыши на форме.

Для того чтобы программа более естественным образом реагировала на действия, описанные в пунктах 1—4, еще раз изменим метод `panel1_MouseUp` (листинг 10.10).

Листинг 10.10. Новый вариант метода panel1_MouseUp

```
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  if (Control.MouseButtons != MouseButtons.None
    && !a.ClientRectangle.Contains(e.Location))
  {
    a.Cursor = null;
    Capture = true;
    return;
  }
  if (Control.MouseButtons == MouseButtons.Left)
    panel1_MouseDown(sender, new MouseEventArgs(MouseButtons.Left,
      0, e.X, e.Y, 0));
  else
  if (Control.MouseButtons == MouseButtons.Right)
    panel1_MouseDown(sender, new MouseEventArgs(MouseButtons.Right,
      0, e.X, e.Y, 0));
  else
    a.Cursor = null;
}
```

Результат. Если теперь выполнить действия 1—4, то положение панели останется неизменным, а курсор мыши будет иметь стандартный вид независимо от того, где он находится.

Комментарии

1. Для достижения описанного результата в методе `panel1_MouseUp` выполняется *программный захват* мыши формой с помощью установки свойства `Capture` формы равным `true`. Программный захват происходит при выполнении двух условий. Во-первых, в этот момент должна оставаться нажатой хотя бы одна кнопка мыши (иначе созданный захват нельзя будет *освободить* корректным образом, отпустив кнопку мыши). Во-вторых, кнопка должна быть отпущена вне компонента, ранее захватившего мышь (для проверки этого факта используется метод `Contains` для свойства `ClientRectangle` компонента, вызвавшего обработчик). Заметим, что свойство `Capture` имеется не только у формы, но и у всех визуальных компонентов.

2. В ряде ситуаций может оказаться полезным событие, сигнализирующее о захвате мыши компонентом или освобождении от захвата. Такое событие предусмотрено для всех визуальных компонентов и имеет имя `MouseCaptureChanged` (оно возникает при изменении значения свойства `Capture` данного компонента). Появилось это событие в .NET 2.0.

ПРИМЕЧАНИЕ

Нельзя сказать, что после сделанных исправлений наша программа будет правильно обрабатывать *все* возможные особые ситуации. Однако теперь, для того чтобы добиться неестественного поведения, потребуются достаточно причудливые действия пользователя. Например, если нажать обе кнопки мыши на свободной части формы, а затем переместить курсор мыши на одну из панелей и отпустить одну из кнопок, то программа перейдет в режим изменения панели, соответствующий той кнопке, которая осталась нажатой. Однако вид курсора при этом не изменится (кроме того, панель не захватит мышь, что повлияет на поведение мыши при ее перемещении за границу обрабатываемой панели). Впрочем, читатель уже получил достаточно информации для самостоятельного исправления подобных недочетов, если такое желание у него возникнет.

10.7. Перетаскивание компонентов любого типа. Поиск и замена в текстах программ

Разместите в форме `Form1` кнопку `button1` и метку `label1` и настройте свойства метки (листинг 10.11). Размеры и положение кнопки и метки установите в соответствии с рис. 10.2.

Свяжите обработчики `panel1_MouseDown`, `panel1_MouseMove` и `panel1_MouseUp` с соответствующими событиями (`MouseDown`, `MouseMove` и `MouseUp`) кнопки `button1` и метки `label1`.

В тексте файла Form1.cs выполните замену слова `Panel` на слово `Control`. Для этого перейдите в начало данного файла и нажмите комбинацию клавиш <Ctrl>+<H>. В появившемся окне **Find and Replace** в поле **Find what** (Найти) введите текст `Panel`, в поле **Replace with** (Заменить на) введите текст `Control` и установите флажок **Match whole word** (Искать слово целиком); если этот флажок в окне не отображается, то нажмите кнопку **Find options** (Опции поиска). После этого нажмите кнопку **Replace all** (Заменить все). После выполнения всех замен будет выведено окно с информацией о том, что заменено 6 найденных слов `Panel` (все эти слова содержатся в трех одинаковых операторах вида `Panel a = sender as Panel`, находящихся в начале методов `panel1_MouseUp`, `panel1_MouseDown` и `panel1_MouseMove`). Для закрытия окна **Find and Replace** нажмите клавишу <Esc>.

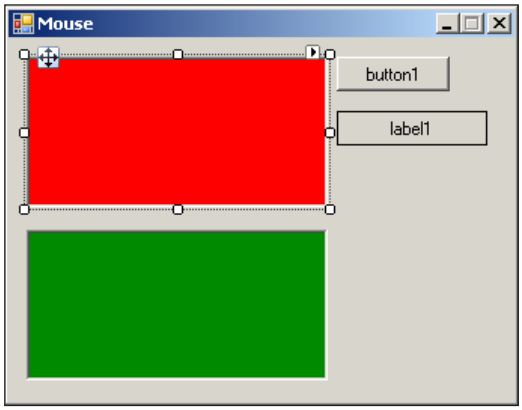


Рис. 10.2. Окончательный вид формы `Form1` для проекта MOUSE

Листинг 10.11. Настройка свойств

```
label1: AutoSize = False, BorderStyle = FixedSingle,
  TextAlign = MiddleCenter
```

Результат. Добавленные компоненты (кнопка `button1` и метка `label1`) обрабатываются так же, как и панели `panel1` и `panel2`: их размер и положение можно изменять с помощью мыши.

Комментарии

1. Методы `panel1_MouseUp`, `panel1_MouseDown` и `panel1_MouseMove` могут обрабатывать события от компонентов различного типа, благодаря приведению параметра `sender` к типу `Control` — общему предку всех визуальных компонентов. Заметим, что в этих методах использованы только те свойства и методы параметра `sender`, которые имеются у класса `Control` (иначе произошла бы ошибка компиляции).
2. Поскольку операции поиска и замены часто оказываются полезными при просмотре и редактировании программного кода, отметим некоторые связанные с ними возможности.

   Для быстрого поиска требуемого фрагмента текста удобно использовать режим **Incremental Search**, устанавливаемый командой меню **Edit | Advanced | Incremental Search** или комбинацией клавиш <Ctrl>+<I>. При установке этого режима курсор мыши принимает вид стрелки, направленной вниз, рядом с которой изображается бинокль. Теперь достаточно напечатать требуемый текстовый фрагмент (вводимый текст отображается на статусной

панели окна Visual C#), и в процессе печати в тексте файла будет выделяться первый из найденных фрагментов, расположенных ниже позиции клавиатурного курсора (т. е. поиск осуществляется *вперед*). Если ввод текстового фрагмента завершен, то для поиска следующего его вхождения далее по тексту достаточно еще раз нажать комбинацию клавиш <Ctrl>+<I>, а для поиска его вхождения назад по тексту — <Ctrl>+<Shift>+<I>. Для выхода из режима **Incremental Search** достаточно нажать клавишу <Esc>. Заметим, что и после выхода из данного режима редактор помнит последний введенный фрагмент для поиска, и если после следующей установки режима **Incremental Search** не набирать новый фрагмент, а сразу нажать комбинацию клавиш <Ctrl>+<I> или <Ctrl>+<Shift>+<I>, то будет выполнен поиск ранее введенного фрагмента текста (в направлении вперед или назад соответственно).

Если при поиске необходимо учитывать дополнительные условия, то следует отобразить диалоговое окно **Find and Replace** в режиме **Quick Find**, нажав комбинацию клавиш <Ctrl>+<F>. Помимо уже упомянутой возможности поиска слова целиком (**Match whole word**) можно также организовать поиск с учетом регистра (**Match case**) и изменить направление поиска на обратное (**Search up**). Если установить флажок **Use** и выбрать из выпадающего списка вариант **Wildcards** (Шаблоны), то во фрагменте для поиска можно указывать следующие шаблоны:

- `*` означает один или более произвольных символов;
- `?` означает ровно один произвольный символ;
- `#` означает ровно одну цифру;
- `[`*символы*`]` означает любой символ из указанного набора;
- `[!`*символы*`]` означает любой символ, не входящий в указанный набор.

Для того чтобы любой из указанных выше символов считался не шаблоном, а обычным символом, перед ним следует указать символ `\` (обратная косая черта). Обычный символ "обратная косая черта" должен указываться как `\\`. Символы-шаблоны можно быстро ввести в поле **Find what**, если воспользоваться кнопкой ▶, расположенной справа от этого поля.

Еще более мощные возможности предоставляет режим поиска с использованием *регулярных выражений* (**Use Regular expressions**), который здесь не рассматривается.

При открытом окне **Find and Replace** для поиска вхождения очередного фрагмента достаточно нажать кнопку **Find Next** или клавишу <Enter>. Поиск можно осуществлять и при закрытом окне, нажимая клавишу <F3>.

Подчеркнем, что нажатие клавиши <F3>, в отличие от комбинации клавиш <Ctrl>+<I>, не требует ввода фрагмента и обеспечивает учет при поиске всех настроек, указанных в окне **Find and Replace**.

Еще одной полезной возможностью, ускоряющей поиск, является выпадающий список ранее искавшихся строк. Этот список находится в верхней панели инструментов, и перейти в него можно комбинацией клавиш <Ctrl>+</>. После этого можно выбрать из списка фрагмент, который искался ранее, или ввести новый текст и нажать клавишу <Enter>. В результате в редакторе будет выделено первое вхождение указанного фрагмента, расположенное далее по тексту. Повторное нажатие клавиши <Enter> выделит следующее вхождение и т. д. (все вхождения будут перебираться циклически). Для возврата в редактор достаточно нажать клавишу <Esc>.

Как уже отмечалось, окно для *поиска и замены* можно отобразить на экране с помощью комбинации клавиш <Ctrl>+<H>. Оно содержит те же флажки настроек, что и окно поиска. Шаблоны в этом окне можно указывать только в строке **Find what**. В режиме поиска и замены можно искать очередное вхождение фрагмента (кнопка **Find Next**), заменять обнаруженное вхождение (кнопка **Replace**, после выполнения замены сразу выполняется поиск следующего вхождения) или заменять все найденные вхождения (кнопка **Replace All**).

Если окно **Find and Replace** является активным, то для его закрытия достаточно нажать клавишу <Esc>.

Глава 11

Перетаскивание (drag & drop): проект ZOO

Проект ZOO знакомит с различными аспектами режима перетаскивания drag & drop (запуск режима drag & drop и различные варианты его завершения, действия при перетаскивании на недопустимый приемник, дополнительное выделение источника и приемника, настройка вида курсора в режиме перетаскивания). Кроме того, описывается работа с компонентом `ImageList` — хранилищем изображений.

11.1. Перетаскивание меток по форме

После создания проекта ZOO разместите в форме `Form1` четыре метки (`label1`—`label4`) и настройте свойства формы и добавленных меток (листинг 11.1). Положение меток настройте в соответствии с рис. 11.1.

Определите обработчик события `MouseDown` для метки `label1` (листинг 11.2), после чего свяжите созданный обработчик с событиями `MouseDown` меток `label2`—`label4` (о связывании обработчиков с несколькими компонентами *см. в разд. 6.1*). Кроме того, определите обработчики событий `DragEnter` и `DragDrop` для формы `Form1` (листинг 11.3).

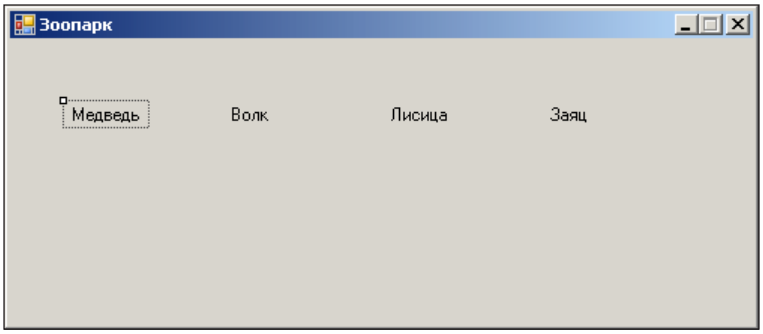


Рис. 11.1. Вид формы `Form1` для проекта ZOO на начальном этапе разработки

Листинг 11.1. Настройка свойств

```
Form1: Text = Зоопарк, MaximizeBox = False, FormBorderStyle = FixedSingle,
  StartPosition = CenterScreen, AllowDrop = True
label1: Text = Медведь
label2: Text = Волк
label3: Text = Лисица
label4: Text = Заяц
```

Листинг 11.2. Обработчик `label1.MouseDown`

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
    DoDragDrop(sender, DragDropEffects.Move);
}
```

Листинг 11.3. Обработчики `Form1.DragEnter` и `Form1.DragDrop`

```
private void Form1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
}
private void Form1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  src.Location = PointToClient(new Point(e.X, e.Y));
}
```

Результат. Метки с названиями зверей можно перетаскивать с помощью левой кнопки мыши. При перетаскивании метки-*источника* (source) она остается на месте, однако вид курсора мыши изменяется, что является признаком режима перетаскивания. В качестве *приемника* (target) пока определена лишь сама форма: при отпускании над ней кнопки мыши происходит перемещение метки зверя на указанную позицию (см. комментарии 1 и 2).

ПРИМЕЧАНИЕ

В данном проекте в текстах обработчиков, связанных с перетаскиванием, будем использовать имена `src` и `trg` для объектов-источников и объектов-приемников соответственно.

Недочет 1. Разрешается перетащить метку не только на свободную часть формы, но и на другую метку; при таком перетаскивании происходит наложение двух меток.

Исправление. Для всех меток установите свойство `AllowDrop` равным **`True`**.

Результат. Теперь при перемещении одной метки на другую курсор мыши принимает вид запрещающего знака (см. комментарий 3).

Недочет 2. В начальный момент перетаскивания курсор принимает вид запрещающего знака.

Исправление. Определите обработчик события `DragEnter` для метки `label1` (листинг 11.4), после чего свяжите созданный обработчик с событиями `DragEnter` меток `label2`—`label4`.

Листинг 11.4. Обработчик `label1.DragEnter`

```
private void label1_DragEnter(object sender, DragEventArgs e)
{
  if (e.Data.GetData(typeof(Label)) == sender)
    e.Effect = DragDropEffects.Move;
  else
    e.Effect = DragDropEffects.None;
}
```

Результат. В начальный момент перетаскивания курсор мыши остается разрешающим. Перетаскивание одной метки на другую по-прежнему запрещено (см. комментарий 4).

Комментарии

1. Для активизации режима перетаскивания любой компонент имеет метод `DoDragDrop`, первый параметр которого указывает *источник* перетаскивания (т. е. тот объект, который будет приниматься приемником), а второй параметр типа `DragDropEffects` — разрешенный результат *(эффект)* успешного перетаскивания. Предусмотрены три основных эффекта: `Copy`, `Move` и `Link` (их названия связаны с вариантами действий при перетаскивании файлов). Если в ходе перетаскивания может возникнуть один из нескольких эффектов (например, `Copy` *или* `Move`), то в качестве второго параметра надо указать все эти эффекты, объединяя их операцией `|` (побитовое ИЛИ), например: `DragDropEffects.Copy | DragDropEffects.Move`. Имеется также член перечисления `DragDropEffects.All`, объединяющий все имеющиеся эффекты.

Приемник перетаскивания может выбрать только тот эффект, который входит в набор разрешенных эффектов, указанных в качестве второго параметра метода DoDragDrop (в обработчике любого события, связанного с перетаскиванием, приемник может определить, какие эффекты разрешены, с помощью свойства e.AllowedEffect). Приемник может также установить эффект None, означающий, что он "отказывается" принимать источник перетаскивания. Эффект, выбранный приемником, определяет вид его курсора в режиме перетаскивания; в частности, если приемник выбрал эффект None, то его курсор будет иметь вид запрещающего знака:  .

Для того чтобы компонент мог выступать в роли приемника, необходимо установить значение его свойства AllowDrop равным **True**. Только в этом случае компонент сможет реагировать на события, связанные с перетаскиванием (в частности, DragEnter, DragOver и DragDrop). Отметим, что обработчик любого из указанных событий имеет параметр e типа DragEventArgs, содержащий информацию о текущем состоянии режима перетаскивания.

Событие DragEnter возникает при *появлении* над компонентом источника перетаскивания, а событие DragOver — при последующем *перемещении* источника над компонентом. В обработчиках этих событий компонент-приемник должен определить, может ли он принять данный источник, и присвоить свойству e.Effect значение соответствующего эффекта. Если при перемещении источника над приемником доступность приемника не может измениться (как в нашей программе), то обработчик события DragOver можно не определять; в подобной ситуации доступность приемника определяется значением свойства e.Effect, заданным в обработчике события DragEnter. Событие DragDrop возникает при *отпускании* источника над компонентом-приемником, причем только в том случае, когда компонент *может* принять источник.

В обработчике любого события, связанного с перетаскиванием, можно получить доступ к объекту-источнику. Для этого надо вызвать метод GetData объекта e.Data, указав в качестве параметра *формат* источника (выражение типа Type или string). Результат, возвращаемый методом GetData, имеет тип object, поэтому его требуется явным образом преобразовать к фактическому типу источника. В ситуации, когда тип источника заранее не известен, приемник может запросить соответствующую информацию, используя метод GetFormats объекта e.Data (без параметров), который возвращает массив строк — имен форматов данных, имеющихся в источнике (поскольку источник перетаскивания может предоставлять данные в нескольких форматах). В нашем случае вызов метода GetFormats вернул бы массив из одного элемента — строки "System.Windows.Forms.Label",

т. е. полного имени типа источника. Заметим, что это имя можно было бы указать в качестве параметра метода `GetData` вместо `typeof(Label)`.

2. Для перемещения метки на новое место необходимо знать позицию курсора мыши при завершении режима перетаскивания. Данная позиция содержится в свойствах `e.X` и `e.Y` (к сожалению, составное свойство `Location` типа `Point` для параметра `e` типа `DragEventArgs` не предусмотрено). При работе со свойствами `X` и `Y` объекта типа `DragEventArgs` следует учитывать, что они определяют позицию курсора мыши в *экранных* координатах. Для перехода от экранных координат к координатам, связанным с клиентской частью визуального компонента (в частности, формы), можно использовать его метод `PointToClient`. Имеется также обратный метод `PointToScreen`, позволяющий перейти от локальных координат компонента к экранным.
3. Установив для формы значение свойства `AllowDrop` равным **`True`**, мы сделали форму допустимым приемником. Поскольку для меток свойство `AllowDrop` оставалось равным **`False`** (значение по умолчанию), метки в первом варианте программы не считались допустимыми приемниками, т. е. они были как бы "невидимы" в режиме перетаскивания. Поэтому, когда курсор мыши в режиме перетаскивания проходил над невидимыми для него компонентами-метками, он считал, что находится над формой, и адресовал события `DragEnter` и `DragDrop` именно форме. Если в этот момент источник отпускался, то он накладывался на расположенную под ним метку (недочет 1). Когда для исправления отмеченного недочета свойства `AllowDrop` меток были установлены в **`True`**, они стали видны в режиме перетаскивания. Так как обработчик `DragEnter` для этих компонентов еще не был определен, считалось, что они не могут принимать источник перетаскивания, поэтому курсор над ними принимал вид запрещающего знака.
4. В условии метода `label1_DragEnter` (листинг 11.4) проверяется, совпадают ли объект-источник (определяемый с помощью метода `GetData`) и объект-приемник (это компонент `sender`, для которого вызван данный обработчик). Обратите внимание на то, что в подобной ситуации не требуется приводить указанные объекты к типу `Label`, так как для сравнения двух объектов на тождество не обязательно знать их фактический тип.

11.2. Перетаскивание меток в поля ввода

Разместите в форме `Form1` четыре поля ввода (`textBox1`—`textBox4`) и настройте их положение в соответствии с рис. 11.2. Для каждого поля ввода положите свойства `ReadOnly` и `AllowDrop` равными **`True`** (настройка `ReadOnly = True`

запрещает пользователю редактировать текст в поле ввода, однако не влияет на возможность программного изменения текста).

Определите обработчики событий `DragEnter` и `DragDrop` для поля ввода `textBox1` (листинг 11.5), после чего свяжите созданные обработчики с событиями `DragEnter` и `DragDrop` полей ввода `textBox2`—`textBox4`.

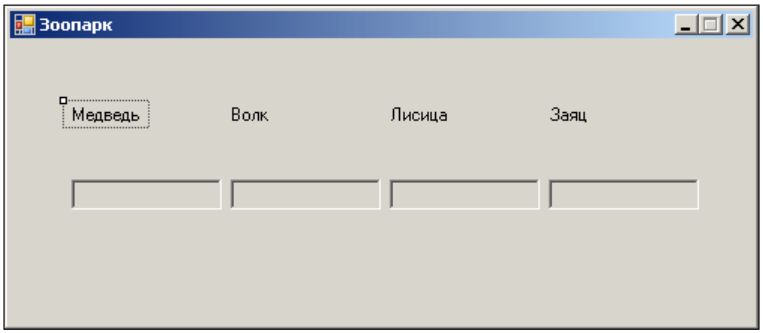


Рис. 11.2. Вид формы `Form1` для проекта ZOO на промежуточном этапе разработки

Листинг 11.5. Обработчики `textBox1.DragEnter` и `textBox1.DragDrop`

```
private void textBox1_DragEnter(object sender, DragEventArgs e)
{
  if ((sender as TextBox).Text == "")
    e.Effect = DragDropEffects.Move;
  else
    e.Effect = DragDropEffects.None;
}
private void textBox1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  (sender as TextBox).Text = src.Text;
  src.Visible = false;
}
```

Результат. Теперь приемником может также служить любое *незаполненное* поле ввода ("пустая клетка"). При перетаскивании метки на незаполненное поле ввода "зверь попадает в клетку" (текст метки отображается в поле ввода). Перетаскивание метки на уже заполненное поле ввода пока запрещено, хотя в следующем разделе это действие станет доступным.

11.3. Взаимодействие меток при их перетаскивании друг на друга

С помощью окна **Properties** установите значения свойства `Tag` для всех компонентов, размещенных в форме: `label1` — **`3`**, `label2` — **`2`**, `label3` — **`1`**, `label4` — **`0`**, `textBox1`—`textBox4` — **`0`**. После установки свойств `Tag` загрузите в редактор файл Form1.Designer.cs (достаточно выполнить двойной щелчок на его имени в окне **Solution Explorer**), разверните в этом файле раздел **Windows Form Designer generated code** и откорректируйте все 8 операторов, задающих значения свойства `Tag`, удалив в них кавычки. Например, текст

```
this.label1.Tag = "3";
```

следует заменить на

```
this.label1.Tag = 3;
```

Измените методы `label1_DragEnter` и `textBox1_DragDrop` (листинг 11.6).

Определите обработчик события `DragDrop` для метки `label1` (листинг 11.7), после чего свяжите созданный обработчик с событиями `DragDrop` меток `label2`—`label4`.

Свяжите обработчик `label1_DragEnter` с событиями `DragEnter` полей ввода `textBox1`—`textBox4`. После выполнения этого действия метод `textBox1_DragEnter` уже не будет связан ни с одним событием, поэтому его описание в файле Form1.cs можно удалить.

Листинг 11.6. Новый вариант методов `label1_DragEnter` и `textBox1_DragDrop`

```
private void label1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
}
private void textBox1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  TextBox trg = sender as TextBox;
  if ((int)src.Tag >= (int)trg.Tag)
  {
    trg.Text = src.Text;
    trg.Tag = src.Tag;
  }
  src.Visible = false;
}
```

Листинг 11.7. Обработчик label1.DragDrop

```
private void label1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  Label trg = sender as Label;
  if ((int)src.Tag > (int)trg.Tag)
  {
    src.Location = trg.Location;
    trg.Visible = false;
  }
  else
    src.Visible = false;
}
```

Результат. При перетаскивании названия одного зверя на название другого более сильный "поедает" более слабого. То же самое происходит, если один из зверей перетаскивается в клетку, уже занятую другим зверем. Подчеркнем, что теперь события `DragEnter` всех компонентов (и меток, и полей ввода) связаны с одним и тем же обработчиком `label1_DragEnter`.

> ***ПРИМЕЧАНИЕ***
>
> Можно было бы связать события `DragEnter` всех компонентов с обработчиком для формы `Form1_DragEnter`, так как он выполняет те же действия. Мы использовали для компонентов отдельный обработчик, поскольку в дальнейшем *(см. разд. 11.5)* собираемся внести в него некоторые изменения.

Ошибка. Если при перетаскивании метки отпустить ее над ней самой, то метка исчезнет. Таким образом, зверь съест самого себя.

Исправление. В методе `label1_DragDrop` (листинг 11.7) перед оператором

```
if ((int)src.Tag > (int)trg.Tag)
```

вставьте

```
if (src == trg) return;
```

Комментарии

1. С помощью свойства `Tag` определяется относительная сила зверей. При помещении зверя в клетку информация о силе зверя сохраняется в свойстве `Tag` клетки (т. е. компонента `TextBox`). Для того чтобы сохранить возмож-

ность помещения зверя в пустую клетку, в начале программы свойства Tag всех полей ввода полагаются равными **0**. В нашем случае в свойствах Tag удобно хранить данные *целого* типа, однако при задании свойства Tag с помощью окна **Properties** ему присваивается строковое значение. Это приводит к необходимости корректировки файла Form1.Designer.cs. Заметим, что после подобной корректировки значения свойств Tag в окне **Properties** отображаются правильно, однако любое их изменение опять приведет к тому, что в Tag будет записана строка, а не число.

2. При определении нового значения для src.Location в методе label1_DragDrop (листинг 11.7) можно было бы использовать параметры e.X и e.Y, как в методе Form1_DragDrop (листинг 11.3), однако проще воспользоваться свойством Location приемника trg. Кроме того, подобный способ позволяет поместить метку-источник в точности на место метки-приемника, независимо от того, в какой точке метки-приемника была отпущена кнопка мыши.

11.4. Действия в случае перетаскивания на недопустимый приемник

Измените метод label1_MouseDown (листинг 11.8).

Листинг 11.8. Новый вариант метода label1_MouseDown

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
    if (DoDragDrop(sender, DragDropEffects.Move) == DragDropEffects.None)
      (sender as Label).Visible = false;
}
```

Результат. Если перетаскивание метки-зверя завершается за пределами формы (в этом случае курсор имеет вид запрещающего знака), то зверь "убегает" из зоопарка и его метка исчезает.

Комментарий

Метод DoDragDrop, активизирующий режим перетаскивания, не возвращает управление вызвавшей его программе до тех пор, пока режим перетаскивания не завершится; при этом он возвращает тот эффект, которым завершилось перетаскивание. В частности, если он вернул значение DragDropEffects.None,

значит, источник был отпущен над недопустимым приемником, т. е. в тот момент, когда курсор мыши имел вид запрещающего знака. В нашей программе все компоненты формы (и сама форма) в любой момент времени являются допустимыми приемниками, поэтому метод `DoDragDrop` вернет значение `DragDropEffects.None` только при отпускании источника за пределами формы.

11.5. Дополнительное выделение источника и приемника в ходе перетаскивания

Измените методы `label1_MouseDown` и `label1_DragEnter` (листинг 11.9).

В методы `label1_DragDrop` и `textBox1_DragDrop` добавьте оператор

```
trg.BackColor = SystemColors.Control;
```

Определите обработчик события `DragLeave` для метки `label1` (листинг 11.10), после чего свяжите созданный обработчик с событиями `DragLeave` меток `label2`—`label4` и полей ввода `textBox1`—`textBox4`.

Листинг 11.9. Новый вариант методов `label1_MouseDown` и `label1_DragEnter`

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
  {
    Label src = sender as Label;
    src.ForeColor = Color.Blue;
    if (DoDragDrop(sender, DragDropEffects.Move) == DragDropEffects.None)
      src.Visible = false;
    src.ForeColor = SystemColors.ControlText;
  }
}
private void label1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
  (sender as Control).BackColor = Color.Yellow;
}
```

Листинг 11.10. Обработчик `label1.DragLeave`

```
private void label1_DragLeave(object sender, EventArgs e)
{
  (sender as Control).BackColor = SystemColors.Control;
}
```

Результат. В режиме перетаскивания цвет текста метки-источника изменяется на синий, а текущий компонент-приемник (метка или поле ввода) изображается на желтом фоне (см. комментарий 1).

Недочет. При выполнении *щелчка* мышью на любой из меток ее фон меняется на желтый.

Исправление. В методе `label1_DragDrop` переместите только что добавленный в него оператор

```
trg.BackColor = SystemColors.Control;
```

на позицию *перед* оператором

```
if (src == trg) return;
```

Результат. Теперь щелчок на метке не приводит к изменению ее внешнего вида (см. комментарий 2).

Комментарии

1. Для дополнительного выделения источника перетаскивания достаточно настроить соответствующим образом его свойства до вызова метода `DoDragDrop`, а после завершения этого метода (т. е. после выхода из режима перетаскивания) восстановить измененные свойства. Для выделения текущего приемника следует изменять его свойства в обработчике события `DragEnter`, а восстанавливать — в обработчике события `DragLeave`, которое возникает в тот момент, когда курсор мыши покидает текущий приемник. Заметим, что в случае недопустимого приемника событие `DragLeave` возникает также в ситуации, когда перетаскивание *завершается* над подобным приемником. Если же перетаскивание завершается над допустимым приемником, то событие `DragLeave` не возникает, и для восстановления свойств приемника надо использовать обработчик события `DragDrop`.

   Обратите внимание на то, что обработчики `label1_DragEnter` и `label1_DragLeave` связываются со *всеми* компонентами формы (и метками, и полями ввода). Это возможно, поскольку класс `Control` — общий предок как меток, так и полей ввода, — имеет свойство `BackColor`, которое требуется

изменить. Таким образом, в этих обработчиках достаточно привести параметр `sender` к типу `Control` (листинги 11.9 и 11.10). В случае обработчика `label1_DragLeave` пригодилось также то обстоятельство, что и метки, и поля ввода в режиме "только для чтения" имеют *одинаковый* стандартный цвет фона `SystemColors.Control`.

2. Отмеченный недочет связан с тем, что при щелчке на кнопке возникает событие `MouseDown`, а обработчик данного события `label1_MouseDown` активизирует режим перетаскивания, который сразу завершается при отпускании кнопки мыши. При этом метка успевает "побыть" приемником, т. е. для нее выполняется событие `DragEnter`, приводящее к изменению фона. Однако в обработчике `label1_DragDrop` фон не восстанавливается, поскольку выход из этого обработчика происходит досрочно в силу выполнения условия `src == trg`. Таким образом, для исправления недочета достаточно восстанавливать цвет фона *перед* проверкой данного условия.

11.6. Настройка вида курсора в режиме перетаскивания

Определите обработчик события `GiveFeedback` для формы `Form1` (листинг 11.11).

Листинг 11.11. Обработчик `Form1.GiveFeedback`

```
private void Form1_GiveFeedback(object sender, GiveFeedbackEventArgs e)
{
  e.UseDefaultCursors = false;
  Cursor.Current = e.Effect == DragDropEffects.Move ?
    Cursors.Hand : Cursors.No;
}
```

Результат. В режиме перетаскивания курсор мыши имеет вид "указывающей руки" (при выходе курсора за границы формы он по-прежнему принимает вид запрещающего знака).

Комментарии

1. Событие `GiveFeedback` класса `Form` специально предназначено для возможности изменения вида курсора в режиме перетаскивания. Определить текущий эффект перетаскивания в обработчике этого события можно с помощью свойства `e.Effect`. В зависимости от эффекта можно установить вид кур-

сора, однако перед этим необходимо отключить отображение стандартных курсоров, предназначенных для перетаскивания, положив свойство `e.UseDefaultCursors` равным `false`. Требуемый курсор надо присвоить статическому свойству `Current` класса `Cursor`.

2. В справочной системе .NET сказано, что статическое свойство `Cursor.Current` позволяет определить и изменить текущий курсор. Это верно лишь отчасти. Действительно, вызов свойства `Cursor.Current` позволяет узнать, как выглядит в данный момент времени курсор в исполняемом приложении. Однако изменить вид курсора с помощью этого свойства можно *только* в обработчике события `MouseMove`. Если же присвоить свойству `Cursor.Current` новое значение в другом месте программы, то это значение будет заменено на значение по умолчанию, как только в программе произойдет событие `MouseMove`. По этой причине данное свойство используется достаточно редко, хотя в обработчике события `GiveFeedback` следует изменять именно свойство `Cursor.Current`. Отметим также, что с помощью свойства `Cursor.Current` можно выявить ситуацию (впрочем, почти невероятную), когда пользователь работает в Windows без манипулятора "мышь" или его заменителя (например, сенсорной панели ноутбука), и курсор мыши на экране отсутствует: в этом случае свойство `Cursor.Current` равно `null`.

11.7. Информация о текущем состоянии программы. Кнопки с изображениями

Разместите в форме `Form1` кнопку `button1`, в качестве ее надписи (т. е. свойства `Text`) укажите **`Зоопарк закрыт`** и настройте положение кнопки в соответствии с рис. 11.3 (для центрирования кнопки `button1` по горизонтали относительно формы можно воспользоваться кнопкой **Center Horizontally** на панели выравнивания **Layout**).

Для большей наглядности добавим слева от надписи кнопки небольшое изображение, вид которого (как и текст кнопки) будет зависеть от текущего состояния программы. Будем использовать рисунки из файлов Critical.bmp и OK.bmp, находящихся в коллекции изображений Visual Studio 2005 (подкаталог Common7\VS2005ImageLibrary\bitmaps\misc). Аналогичные файлы имеются в коллекции Visual Studio 2008 (подкаталог Common7\VS2008ImageLibrary\Annotations&Buttons\bmp_format). Если указанные коллекции недоступны, то можно использовать любые растровые рисунки, имеющие размер 16 × 16 пикселов. Добавьте в форму компонент типа `ImageList`, предназначенный для хранения набора изображений одинакового размера (этому компоненту будет

присвоено имя imageList1; он будет располагаться в области невизуальных компонентов под изображением формы Form1). Для добавления к компоненту imageList1 рисунков Critical.bmp и OK.bmp выполните следующие действия:

1. Выберите в окне свойств компонента imageList1 свойство Images и нажмите кнопку с многоточием [...] рядом с этим свойством.
2. В появившемся окне **Images Collection Editor** нажмите кнопку **Add** и выберите файл Critical.bmp в появившемся окне открытия файла (в результате файл Critical.bmp будет добавлен в коллекцию Images).
3. Аналогичными действиями добавьте в коллекцию рисунок OK.bmp, после чего закройте окно **Images Collection Editor**, нажав клавишу <Enter> или кнопку **OK**.

Настройте еще несколько свойств компонентов imageList1 и button1 (листинг 11.12; при настройке свойства ImageAlign компонента button1 надо выбрать в появившейся панели квадрат, расположенный в левом нижнем углу).

Измените метод label1_MouseDown, добавив к нему новые операторы (листинг 11.13).

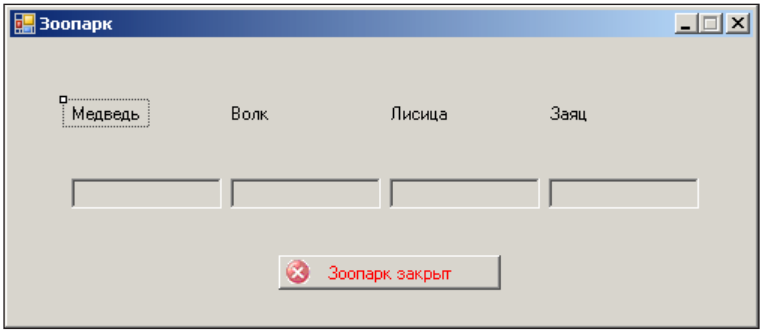


Рис. 11.3. Окончательный вид формы Form1 для проекта ZOO

Листинг 11.12. Настройка свойств

```
imageList1: TransparentColor = Magenta
button1: ForeColor = Red, ImageList = imageList1,
  ImageKey = Critical.bmp, ImageAlign = BottomLeft
```

Листинг 11.13. Добавление к методу label1_MouseDown

```
string s = "";
for (int i = 1; i <= 4; i++)
{
```

```
  if (Controls["label" + i].Visible)
  // один из зверей на свободе
    return;
  s = s + (Controls["textBox" + i].Text);
}
if (s == "")
  // все клетки пусты
  return;
button1.Text = "Зоопарк открыт";
button1.ForeColor = Color.Green;
button1.ImageKey = "OK.bmp";
```

Результат. Если все метки исчезли и при этом хотя бы одно поле ввода оказалось заполненным, то на кнопке выводится текст **Зоопарк открыт**, а цвет оформления кнопки меняется с красного на зеленый (см. комментарии 1 и 2).

> ***ПРИМЕЧАНИЕ***
>
> Для перебора всех меток и всех полей ввода в новом варианте метода `label1_MouseDown` использовалось свойство-коллекция `Controls` формы.

Недочет. Перетаскивание метки на кнопку `button1` приводит к тому, что метка "проваливается" под кнопку и становится недоступной для последующего перетаскивания, поскольку ее не удается зацепить мышью.

Исправление. Положите свойство `AllowDrop` кнопки `button1` равным **`True`** и свяжите с событием `DragEnter` кнопки обработчик **`Form1_DragEnter`**.

Результат. Теперь при попытке перетаскивания метки на кнопку `button1` ничего не происходит: метка остается на прежнем месте (см. комментарий 3).

Комментарии

1. Следует обратить внимание на свойство `TransparentColor` компонента `ImageList`. В коллекциях изображений, предназначенных для размещения на кнопках, один из цветов (обычно очень яркий и поэтому не используемый) применяется в качестве индикатора прозрачного цвета: этим цветом помечаются те фрагменты рисунка, сквозь которые должен просвечивать компонент, содержащий данный рисунок. В коллекции рисунков, входящей в Visual Studio 2005 и 2008, в качестве такого цвета выбран фиолетовый (R = 255, G = 0, B = 255; такой цвет в перечислении `KnownColors` представлен двумя элементами-синонимами: `Magenta` и `Fuchsia`). Если для свойства `TransparentColor`

оставить значение **Transparent**, установленное по умолчанию, то по краям рисунков на кнопках будут изображаться некрасивые фиолетовые фрагменты.

2. Для указания того, какой рисунок из набора `ImageList` надо подключить к данному компоненту, можно использовать не только строковое свойство `ImageKey`, содержащее имя рисунка (как в последнем операторе листинга 11.13), но и целочисленное свойство `ImageIndex`, содержащее индекс рисунка в наборе (элементы набора индексируются от 0). Так, подключить рисунок OK.bmp к кнопке `button1` можно было бы с помощью оператора `button1.ImageImdex = 1`. Индексы удобно использовать при подключении различных изображений к нескольким компонентам в цикле.
3. Отмеченный недочет обусловлен тем, что по умолчанию свойство `AllowDrop` кнопки равно **`False`**, и поэтому кнопка невидима для режима перетаскивания. Заметим, что простого изменения значения свойства `AllowDrop` на **`True`** недостаточно: в этом случае кнопка будет вести себя как *недопустимый* приемник, и поэтому метка, отпущенная над кнопкой, будет исчезать с формы, поскольку именно такое действие выполняется в нашей программе при попытке отпустить источник над недопустимым приемником *(см. разд. 11.4)*. Связывание события `DragEnter` кнопки с обработчиком `Form1_DragEnter` решает проблему, так как в этом случае кнопка становится допустимым приемником с эффектом `Move`, хотя при отпускании над ней источника она ничего не делает (последнее обстоятельство связано с тем, что для кнопки не определен обработчик события `DragDrop`).

11.8. Восстановление исходного состояния

Определите обработчики события `Load` для формы `Form1` и события `Click` для кнопки `button1` (листинг 11.14).

Листинг 11.14. Обработчики `Form1.Load` и `button1.Click`

```
private void Form1_Load(object sender, EventArgs e)
{
  for (int i = 1; i <= 4; i++)
    Controls["label" + i].Location =
      Controls["textBox" + i].Location - new Size(0, textBox1.Top / 2);
  ActiveControl = button1;
}
```

```
private void button1_Click(object sender, EventArgs e)
{
  Form1_Load(this, null);
  for (int i = 1; i <= 4; i++)
  {
    Controls["label" + i].Visible = true;
    Control c = Controls["textBox" + i];
    c.Text = "";
    c.Tag = 0;
  }
  button1.Text = "Зоопарк закрыт";
  button1.ForeColor = Color.Red;
  button1.ImageKey = "Critical.bmp";
}
```

Результат. Исходное положение меток-зверей теперь определяется программно, а именно в обработчике события формы `Load` (метки располагаются над соответствующими полями ввода и выравниваются по их левой границе). В дальнейшем при нажатии кнопки `button1` исходное положение зверей восстанавливается, а клетки освобождаются. Кроме того, при запуске программы кнопка `button1` становится активной, что приводит к появлению вокруг нее рамки красного цвета (цвет рамки вокруг активной кнопки определяется цветом `ForeColor` ее надписи).

> ***ПРИМЕЧАНИЕ***
>
> Обратите внимание на то, что в методах, описанных в листинге 11.14, компоненты, получаемые из коллекции `Controls`, не приводятся к их фактическому типу (`Label` или `TextBox`). В этом нет необходимости, поскольку все используемые в этих методах свойства компонентов имеются уже у класса `Control` — общего предка всех визуальных компонентов, а коллекция `Controls` возвращает элементы типа `Control`.

Комментарий

Событие `Load` формы возникает единственный раз: после выполнения конструктора формы, но до отображения формы на экране. Это событие особенно полезно, если возможны ситуации, когда после запуска программы вместо отображения формы надо вывести какое-либо сообщение об ошибке и немедленно завершить программу (приведем пример такой ситуации: демонстраци-

онная программа обнаружила, что истек срок, отведенный для ознакомления с ее возможностями; *см. также разд. 30.5*). Хотя описанные ситуации можно выявить уже в конструкторе формы, вызывать из конструктора формы метод `Close` нельзя, так как это приведет к ошибке времени выполнения. Для того чтобы корректно завершить программу до появления формы на экране, следует вызвать метод `Close` в обработчике события `Load`.

Помещать фрагмент кода, выполняющий какие-либо инициализирующие действия, в обработчик события `Load` (а не в конструктор формы) имеет смысл также в случае, если программе в дальнейшем может потребоваться повторно выполнять эти действия. В подобной ситуации достаточно явным образом вызвать обработчик события `Load`, как это сделано в методе `button1_Click` (листинг 11.14).

Глава 12

Работа с датами и временем: проект CLOCK

Проект CLOCK посвящен типам, связанным с датами и временем (структуры `DateTime`, `TimeSpan` и невизуальный компонент `Timer`). Рассматриваются два варианта реализации секундомера и описываются действия, обеспечивающие отображение часов и секундомера на панели задач при сворачивании окна приложения.

12.1. Отображение текущего времени

После создания проекта CLOCK добавьте в форму `Form1` метку `label1`, а также невизуальный компонент типа `Timer` (этот компонент получит имя `timer1` и будет размещен ниже формы, в области невизуальных компонентов). Настройте свойства формы и добавленных компонентов (листинг 12.1). При задании свойств метки `label1` обратите внимание на ее свойство `Font`, также имеющее набор свойств, два из которых — `Name` и `Size` — надо изменить (в листинге 12.1 для этих свойств используется точечная нотация: `Font.Name` и `Font.Size`). Положение метки `label1` настройте в соответствии с рис. 12.1.

Определите обработчик события `Tick` для компонента `timer1` (листинг 12.2).

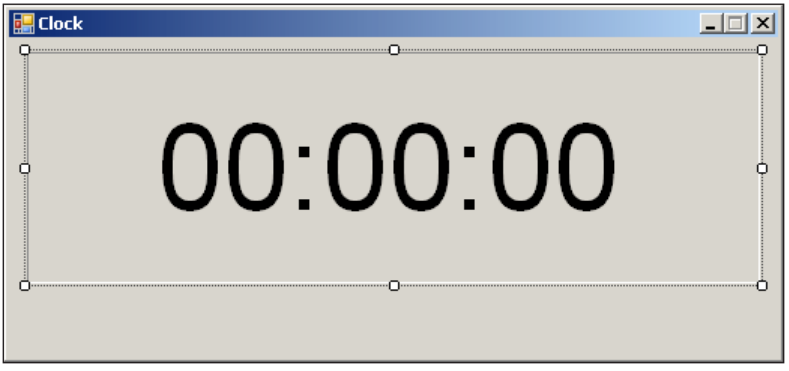


Рис. 12.1. Вид формы `Form1` для проекта CLOCK на начальном этапе разработки

Листинг 12.1. Настройка свойств

```
Form1: Text = Clock, MaximizeBox = False, FormBorderStyle = FixedSingle,
  StartPosition = CenterScreen
label1: Text = 00:00:00, AutoSize = False, TextAlign = MiddleCenter,
  BorderStyle = Fixed3D, Font.Name = Arial, Font.Size = 60
timer1: Enabled = True, Interval = 1000
```

Листинг 12.2. Обработчик `timer1.Tick`

```
private void timer1_Tick(object sender, EventArgs e)
{
  label1.Text = DateTime.Now.ToLongTimeString();
}
```

Результат. При работе программы в ее окне отображается текущее время.

Недочет. В течение первой секунды после запуска в окне программы сохраняется исходный текст метки, так как событие `Tick` возникает первый раз только через промежуток времени `timer1.Interval`, равный в нашем случае `1000` (время указывается в миллисекундах).

Исправление. Свяжите обработчик `timer1_Click` с событием `Load` формы `Form1`.

Результат. Теперь метод `timer1_Click` выполняется первый раз непосредственно перед отображением формы на экране, поэтому в окне сразу выводится правильное время.

Комментарий

Для работы с датами и временем в библиотеке классов .NET Framework предусмотрена структура `DateTime`. Ее статическое свойство `Now`, доступное только для чтения, возвращает текущую дату и время (по системным часам компьютера). Только текущую дату (время соответствует полуночи) можно получить с помощью статического свойства `Today`. Для преобразования даты/времени к их стандартным строковым представлениям можно использовать следующие методы структуры `DateTime`:

- `ToShortDateString` — дата в кратком формате `d`, например, `27.01.1756`;
- `ToLongDateString` — дата в полном формате `D`, например, `27 января 1756 г.`;

- `ToShortTimeString` — время в кратком формате `t`, например, `10:55`;
- `ToLongTimeString` — время в полном формате `T`, например, `10:55:15`.

Метод `ToString` без параметров возвращает дату/время в формате `G` (дата в кратком формате, время — в полном). Формат отображения даты/времени можно явно указать в методе `ToString`. Например, в нашей программе можно было бы использовать такой вариант: `DateTime.Now.ToString("T")`.

Упомянем еще некоторые форматы для даты/времени: `g` — дата и время в кратком формате, `F` — дата и время в полном формате, `f` — дата в полном формате, время — в кратком, `M` или `m` — формат "месяц, день", `Y` или `y` — формат "месяц, год".

При форматировании дат используются текущие *региональные настройки* (в нашем случае — настройки для России), хотя имеется перегруженный вариант метода `ToString`, в котором можно явно указать требуемую региональную настройку. Можно также сменить региональную настройку для приложения в целом; для этого достаточно установить новое значение свойства `Application.CurrentCulture` (ранее мы использовали это свойство в *разд. 6.4* для получения информации о текущих региональных настройках). Например, установить для приложения региональные настройки, соответствующие американскому варианту английского языка, можно с помощью оператора

```
Application.CurrentCulture =
  new System.Globalization.CultureInfo("en-US");
```

Заметим, что настройки для России имеют имя `ru-RU`.

12.2. Реализация возможностей секундомера

Разместите в форме `Form1` компонент-*флажок* типа `CheckBox` (он получит имя `checkBox1`) и две кнопки (`button1` и `button2`) и настройте их свойства (листинг 12.3). Положение добавленных компонентов настройте в соответствии с рис. 12.2; для горизонтального центрирования добавленной группы компонентов выделите эти компоненты и нажмите кнопку **Center Horizontally** на панели **Layout**.

В описание класса `Form1` добавьте описание поля:

```
private int t;
```

В начало метода `timer1_Tick` добавьте новые операторы (листинг 12.4).

Определите обработчик события `CheckedChanged` для флажка `checkBox1` и обработчики события `Click` для кнопок `button1` и `button2` (листинг 12.5).

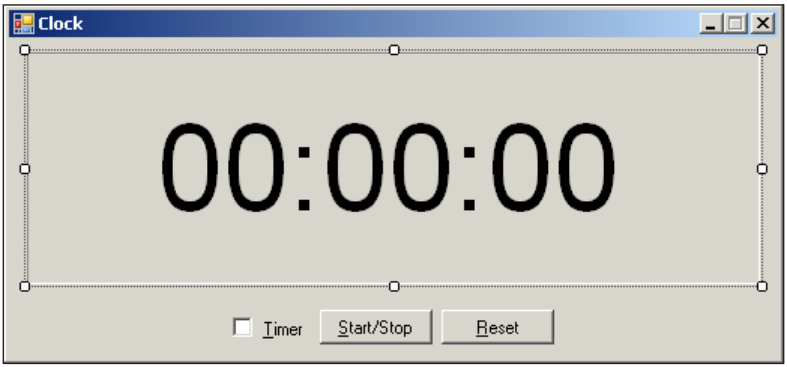


Рис. 12.2. Окончательный вид формы `Form1` для проекта CLOCK

Листинг 12.3. Настройка свойств

```
checkBox1: Text = &Timer
button1: Text = &Start/Stop, Enabled = False
button2: Text = &Reset, Enabled = False
```

Листинг 12.4. Новый вариант метода `timer1_Tick`

```
private void timer1_Tick(object sender, EventArgs e)
{
  if (checkBox1.Checked)
  {
    t++;
    label1.Text = string.Format("Timer: {0}:{1}", t / 10, t % 10);
  }
  else
    label1.Text = DateTime.Now.ToLongTimeString();
}
```

Листинг 12.5. Обработчики `checkBox1.CheckedChanged`, `button1.Click` и `button2.Click`

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
  if (checkBox1.Checked)
  {
```

```
    t = -1;
    timer1.Interval = 100;
  }
  else
    timer1.Interval = 1000;
  timer1_Tick(this, null);
  button1.Enabled = button2.Enabled = checkBox1.Checked;
  timer1.Enabled = true;
}
private void button1_Click(object sender, EventArgs e)
{
  timer1.Enabled = !timer1.Enabled;
}
private void button2_Click(object sender, EventArgs e)
{
  timer1.Enabled = false;
  t = 0;
  label1.Text = "Timer: 0:0";
}
```

Результат. При установке флажка **Timer** во включенное состояние программа переходит в режим секундомера, причем секундомер сразу запускается, отображая на экране число прошедших с момента запуска секунд с точностью до десятой доли секунды. Запуск и остановка секундомера осуществляются по нажатию кнопки **Start/Stop**, сброс секундомера — по нажатию кнопки **Reset**. Доступны клавиши-ускорители: <Alt>+<T> (смена режима "часы/секундомер"), <Alt>+<S> (запуск/остановка секундомера), <Alt>+<R> (сброс секундомера).

Ошибка. Кажущаяся правильность работы секундомера обманчива. В этом можно убедиться, если не останавливать секундомер в течение некоторого времени (выполняя при этом другие действия на компьютере), после чего сравнить результат с точным временем. Причина заключается в том, что событие `Tick` в режиме секундомера происходит *примерно* каждые 100 мс; кроме того, надо учитывать, что событие `Tick` наступает только при отсутствии других событий, которые требуется обработать программе. Для правильной реализации секундомера надо связать его с *часами компьютера* (используя метод `Now`).

Исправление. В описании класса `Form1` замените строку

```
private int t;
```

на описание новых полей:

```
private DateTime startTime, pauseTime;
private TimeSpan pauseSpan;
```

Назначение добавленных полей следующее: поле `startTime` содержит время начального запуска секундомера; поле `pauseTime` содержит время последней остановки секундомера, а поле `pauseSpan` содержит суммарную длительность всех остановок, выполненных после начального запуска.

Измените методы `timer1_Tick`, `checkBox1_CheckedChanged`, `button1_Click` и `button2_Click` (листинг 12.6).

Листинг 12.6. Новые варианты методов `timer1_Tick`, `checkBox1_CheckedChanged`, `button1_Click` и `button2_Click`

```
private void timer1_Tick(object sender, EventArgs e)
{
  if (checkBox1.Checked)
  {
    TimeSpan s = DateTime.Now - startTime - pauseSpan;
    label1.Text = string.Format("Timer: {0}:{1}",
      s.Minutes * 60 + s.Seconds, s.Milliseconds / 100);
  }
  else
    label1.Text = DateTime.Now.ToLongTimeString();
}
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
  if (checkBox1.Checked)
  {
    startTime = DateTime.Now;
    pauseSpan = TimeSpan.Zero;
    timer1.Interval = 100;
  }
  else
    timer1.Interval = 1000;
  timer1_Tick(this, null);
  button1.Enabled = button2.Enabled = checkBox1.Checked;
  timer1.Enabled = true;
```

```
}
private void button1_Click(object sender, EventArgs e)
{
  timer1.Enabled = !timer1.Enabled;
  if (timer1.Enabled)
    pauseSpan += DateTime.Now - pauseTime;
  else
    pauseTime = DateTime.Now;
}
private void button2_Click(object sender, EventArgs e)
{
  timer1.Enabled = false;
  pauseTime = startTime;
  pauseSpan = TimeSpan.Zero;
  label1.Text = "Timer: 0:0";
}
```

Комментарий

Для хранения *относительных* промежутков времени (как положительных, так и отрицательных) предназначена структура `TimeSpan`. Промежутки времени измеряются в днях, часах, минутах, секундах и миллисекундах, причем для получения значения каждого из этих компонентов можно использовать соответствующие свойства структуры `TimeSpan`: `Days`, `Hours`, `Minutes`, `Seconds`, `Milliseconds` (заметим, что для выделения одного из компонентов структуры `DateTime` надо использовать ее свойства `Day`, `Hour`, `Minute`, `Second`, `Millisecond`, а также `Year` и `Month`). Для задания нулевого промежутка времени проще всего воспользоваться полем `TimeSpan.Zero`, доступным только для чтения. И структура `DateTime`, и структура `TimeSpan` имеют также поля для чтения, определяющие их наименьшие и наибольшие возможные значения: `MinValue` и `MaxValue`.

Операции сложения и вычитания определяются для структур `DateTime` и `TimeSpan` следующим образом: сумма или разность значений типа `TimeSpan` имеет тип `TimeSpan`; сумма или разность значений типа `DateTime` и `TimeSpan` *(в указанном порядке)* имеет тип `DateTime`; разность значений типа `DateTime` имеет тип `TimeSpan`; складывать значения типа `DateTime` нельзя. Поскольку для промежутков времени допускаются отрицательные значения, для структуры `TimeSpan` определена также операция "унарный минус".

Для создания объектов типа DateTime и TimeSpan с требуемыми значениями проще всего воспользоваться одним из предусмотренных для них конструкторов. Конструктор без параметров возвращает для DateTime минимальную дату (полночь 1 января 1 года н. э.), а для TimeSpan — нулевой промежуток времени. В остальных вариантах конструктора DateTime необходимо указывать год, месяц, день (и можно указать дополнительно время в часах, минутах и секундах). Время, указанное в конструкторе DateTime, можно дополнить числом миллисекунд. В конструкторах TimeSpan необходимо указывать час, минуту и секунду; в качестве дополнительного *начального* параметра можно указать число дней. Если в конструкторе TimeSpan указано число дней, то можно указать дополнительный *последний* параметр — число миллисекунд.

12.3. Альтернативные варианты выполнения команд с помощью мыши

Определите обработчик события MouseDown для метки label1 (листинг 12.7).

Листинг 12.7. Обработчик label1.MouseDown

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Clicks == 2)
    checkBox1.Checked = !checkBox1.Checked;
  else
  {
    if (!button1.Enabled) return;
    if (e.Button == MouseButtons.Left)
      button1_Click(this, null);
    else
      if (e.Button == MouseButtons.Right)
        button2_Click(this, null);
  }
}
```

Результат. Двойной щелчок любой кнопкой мыши на метке приводит к смене режима "часы/секундомер", одинарный щелчок левой кнопкой в режиме секундомера запускает или останавливает секундомер, одинарный щелчок правой кнопкой мыши приводит к сбросу секундомера.

Комментарий

1. Объединить в одном обработчике действия при одинарном и двойном щелчке мышью нам удалось, благодаря наличию у параметра `e` (типа `MouseEventArgs`) свойства `Clicks`, которое может принимать значение `1` или `2`.
2. При связывании некоторых действий с одинарным и двойным щелчком мыши необходимо учитывать, что при выполнении двойного щелчка система вначале регистрирует одинарный щелчок (при котором наступает событие `MouseDown` с `e.Clicks`, равным `1`, затем событие `Click`, затем `MouseUp` с `e.Clicks`, равным `1`), и только потом, после второго нажатия кнопки мыши, если промежуток времени между нажатиями был достаточно мал, регистрируется двойной щелчок (при котором наступают события `MouseDown` с `e.Clicks`, равным `2`, затем `DoubleClick`, затем `MouseUp` с `e.Clicks`, равным `2`). Поэтому необходимо, чтобы действие, выполняющееся при одинарном щелчке, не конфликтовало с действием, которое связывается с двойным щелчком. В нашей программе в этом отношении все обстоит благополучно: хотя в режиме секундомера предусмотрены действия и при одинарном, и при двойном щелчке, действие при одинарном щелчке (запуск, или остановка, или сброс секундомера) никак не конфликтует с действием, связанным с двойным щелчком (т. е. с переходом в режим часов).

12.4. Отображение текущего состояния часов и секундомера на панели задач

В метод `timer1_Tick` добавьте новый оператор:

```
Text = WindowState == FormWindowState.Minimized ?
  label1.Text : "Clock";
```

Результат. Если минимизировать окно приложения CLOCK, то на его кнопке, расположенной на панели задач у нижней границы экрана, будет отображаться, в зависимости от режима, текущее время или данные секундомера с пояснением **Timer**. Если окно приложения находится в обычном состоянии, то на кнопке приложения отображается текст, совпадающий с заголовком окна: **Clock.**

Недочет. Если минимизировать окно в режиме остановленного секундомера, то текст кнопки приложения не изменится.

Исправление. Определите обработчик события `Resize` для формы `Form1` (листинг 12.8).

Листинг 12.8. Обработчик `Form1.Resize`

```
private void Form1_Resize(object sender, EventArgs e)
{
  Text = WindowState == FormWindowState.Minimized ?
    label1.Text : "Clock";
}
```

Результат. Теперь текст на кнопке приложения правильно корректируется в любой ситуации, поскольку событие `Resize` происходит не только при изменении размера формы, но и при ее минимизации и возврате в обычное состояние. Кроме того, корректировка текста на кнопке приложения теперь происходит быстрее (до сделанного исправления текст на кнопке изменялся только при очередном срабатывании таймера, а в режиме часов до этого события могла пройти целая секунда).

ПРИМЕЧАНИЕ

При определении обработчика для события формы `Resize` возникает соблазн связать это событие с уже имеющимся обработчиком `timer1_Tick`, что позволило бы не дублировать одинаковый текст в программе. К сожалению, подобный вариант исправления тоже неправильно работает в режиме остановленного секундомера (объясните почему). Вместе с тем, избежать дублирования текста все же можно, если, наоборот, в обработчике `timer1_Tick` вместо добавленного в данном разделе текста поместить вызов метода `Form1_Resize`: **`Form1_Resize(this, null);`**

Глава 13

Просмотр изображений: проект IMGVIEW

Проект IMGVIEW посвящен компонентам `DriveListBox`, `DirListBox` и `FileListBox`, обеспечивающим визуализацию файловой структуры компьютера. Перед применением этих компонентов описываются действия, необходимые для их размещения в окне **Toolbox**. Кроме того, дается описание компонентов `SplitContainer` и `PictureBox` и рассматриваются вопросы, связанные со стыковкой компонентов (docking) и использованием реестра Windows для хранения информации о текущих настройках приложения (классы `Registry` и `RegistryKey`).

13.1. Добавление в окно *Toolbox* новых компонентов

После создания проекта IMGVIEW установите свойство `Text` формы `Form1` равным **`Image Viewer`**, а ее свойство `StartPosition` равным **`CenterScreen`**.

Перед размещением в форме новых компонентов добавим в окно **Toolbox** три компонента, обеспечивающих просмотр элементов файловой системы компьютера: `DriveListBox` (просмотр и выбор дисков), `DirListBox` (просмотр и выбор каталогов, находящихся на указанном диске) и `FileListBox` (просмотр и выбор файлов, находящихся в указанном каталоге).

Для добавления этих компонентов в окно **Toolbox** перейдите в группу компонентов, в которую следует поместить новые компоненты (например, **All Windows Forms**), вызовите ее контекстное меню, щелкнув правой кнопкой мыши, и выполните команду **Choose Items...**. Данная команда может выполняться достаточно долгое время, поскольку в ходе ее выполнения формируется список всех компонентов, доступных в .NET Framework. После завершения формирования списка доступных компонентов он отображается на экране в окне **Choose Toolbox Items**. На вкладке **.NET Framework Components** этого окна установите флажки рядом с именами компонентов `DriveListBox`, `DirListBox` и `FileListBox`, после чего нажмите клавишу <Enter> или кнопку **OK**.

Результат. Отмеченные компоненты добавлены в текущую группу в окне **Toolbox**. С каждым из добавленных компонентов связывается одинаковое изображение в виде шестеренки:  .

Комментарий

Компоненты `DriveListBox`, `DirListBox` и `FileListBox` включены в библиотеку .NET для возможности быстрого переноса в среду .NET программ, разработанных в среде Visual Basic версий 5 и 6, поскольку в этих версиях они имеются и активно используются. Поэтому в справочной системе .NET информация об этих компонентах содержится только в разделе, связанном с языком Visual Basic. Однако платформа .NET обеспечивает настолько тесную интеграцию различных языков программирования, что никаких проблем при использовании данных компонентов в программах на Visual C# не возникает.

Разумеется, в библиотеке классов .NET имеются и другие средства для организации просмотра файловой системы, и в некоторых отношениях они имеют преимущество перед компонентами-списками, перенесенными из среды Visual Basic. Примерами таких средств являются диалоговые окна `OpenFileDialog` и `SaveFileDialog` *(см. разд. 14.1, 18.2 и 18.3)*, а также `FolderBrowserDialog`. Однако простота использования файловых компонентов-списков и возможность их встраивания непосредственно в форму делает их по-прежнему востребованными при разработке Windows-приложений.

13.2. Просмотр изображений из файлов на текущем диске

Разместите в форме `Form1` компонент типа `SplitContainer` (будем называть его *split-контейнером*). Этот компонент получит по умолчанию имя `splitContainer1` и сразу займет всю свободную часть формы (поскольку его свойство `Dock` по умолчанию равно **`Fill`**). Компонент `splitContainer1` состоит из двух дочерних панелей (левой с именем `Panel1` и правой с именем `Panel2`); между этими панелями находится *разделитель* (splitter) — область, зацепив за которую можно увеличить ширину одной панели за счет уменьшения ширины другой. Каждую из дочерних панелей можно выделить; при этом в окне **Properties** отображаются свойства выделенной панели. Разделитель выделить нельзя; при щелчке на нем мышью выделяется весь компонент `splitContainer1` (заметим, что это наиболее удобный способ выделения split-контейнера, так как остальная его часть занята дочерними панелями).

Разместите на панели `Panel2` компонента `splitContainer1` еще один split-контейнер (он получит имя `splitContainer2`).

Хотя второй split-контейнер помещен на одну из панелей первого split-контейнера, визуально форма будет содержать три равноправные панели одинаковой высоты, разделенные одинаковыми по ширине разделителями. Определить, к какому контейнеру принадлежит та или иная панель, можно с помощью верхней части окна **Properties**. При последовательном выделении панелей формы слева направо в верхней части окна **Properties** будут указаны следующие имена: `splitContainer1.Panel1`, `splitContainer2.Panel1` и `splitContainer2.Panel2`. Заметим, что щелчок мышью на разделителе, расположенном между двумя левыми панелями, приведет к выделению компонента `splitContainer1`, а щелчок на разделителе между двумя правыми панелями — к выделению компонента `splitContainer2`.

Теперь заполним панели split-контейнеров реальным содержимым. На панель `Panel1` компонента `splitContainer1` поместите компонент типа `DirListBox`, на панель `Panel1` компонента `splitContainer2` поместите компонент типа `FileListBox`, на панель `Panel2` компонента `splitContainer2` поместите компонент типа `PictureBox` (добавленные компоненты получат имена `dirListBox1`, `fileListBox1` и `pictureBox1`). Настройте свойства добавленных компонентов (листинг 13.1; рис. 13.1). Напомним, что при настройке свойства `Dock` в окне **Properties** отображается специальная панель; для выбора в этой панели требуемого варианта **`Fill`** надо щелкнуть мышью на центральном прямоугольнике.

Определите обработчики события `Change` для компонента `dirListBox1` и события `SelectedIndexChanged` для компонента `fileListBox1` (листинг 13.2).

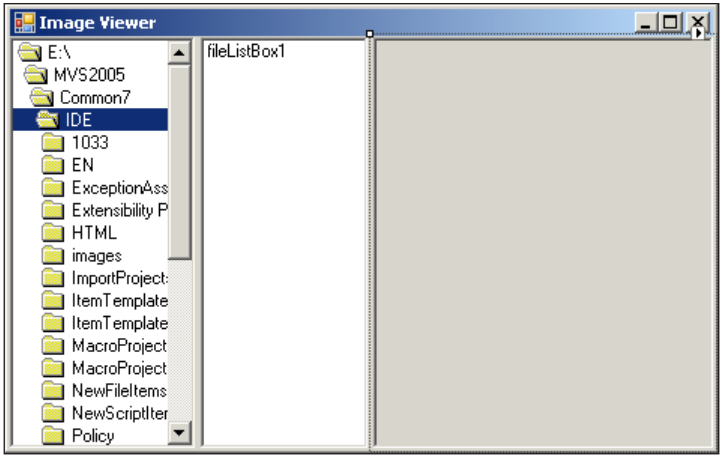


Рис. 13.1. Вид формы `Form1` для проекта IMGVIEW на начальном этапе разработки

Листинг 13.1. Настройка свойств

```
dirListBox1: Dock = Fill
fileListBox1: Dock = Fill, IntegralHeight = False,
  Pattern = *.bmp;*.jpg;*.png;*.gif;*.ico;*.wmf;*.emf
pictureBox1: Dock = Fill, SizeMode = Zoom, BorderStyle = Fixed3D
```

Листинг 13.2. Обработчики `dirListBox1.Change` и `fileListBox1.SelectedIndexChanged`

```
private void dirListBox1_Change(object sender, EventArgs e)
{
  fileListBox1.Path = dirListBox1.Path;
}
private void fileListBox1_SelectedIndexChanged(object sender,
  EventArgs e)
{
  try
  {
    pictureBox1.Image = new Bitmap(fileListBox1.Path +
      "\\" + fileListBox1.FileName);
  }
  catch
  {
    pictureBox1.Image = null;
  }
}
```

Результат. Программа позволяет перемещаться по каталогам текущего диска. При запуске программы выбирается ее *рабочий каталог*; по умолчанию этим каталогом считается каталог, из которого запущен exe-файл (в нашем случае рабочим каталогом будет подкаталог bin\Debug каталога с проектом IMGVIEW, поэтому для ускорения тестирования проекта IMGVIEW в его подкаталог bin\Debug целесообразно скопировать несколько графических файлов разных типов).

Для выбора нового каталога требуется выполнить двойной щелчок мышью на его имени в списке каталогов; при этом в списке каталогов дополнительно отображаются имена всех подкаталогов выбранного каталога, а в списке файлов

выводятся имена графических файлов (с расширениями bmp, jpg, png, gif, ico, wmf, emf), находящихся в выбранном каталоге. При щелчке на имени файла (или при перемещении по списку файлов с помощью клавиш со стрелками) на правой панели в компоненте `pictureBox1` выводится изображение из выбранного файла; при этом изображение масштабируется по размерам панели с сохранением его исходных пропорций.

Ширину панелей можно изменять путем перетаскивания мышью разделителей, расположенных между ними (при перемещении курсора мыши на эти разделители он принимает вид двунаправленной стрелки). Можно также выделить требуемый разделитель, используя клавишу <Tab>, после чего переместить его влево или вправо с помощью клавиш со стрелками. При изменении ширины формы происходит пропорциональное изменение ширины каждой из трех панелей. См. также комментарии 1—4.

Недочет 1. При изменении размеров формы (и происходящем при этом синхронном изменении ширины ее панелей) содержимое списка каталогов и списка файлов неприятно мерцает в результате многократной перерисовки.

Исправление. Для компонентов `splitContainer1` и `splitContainer2` положите свойство `FixedPanel` равным **`Panel1`**.

Результат. Теперь изменение ширины формы сказывается только на правой панели, содержащей изображение; ширина панелей со списками каталогов и файлов не изменяется. В подобной ситуации мерцание списков практически исчезает (мерцает лишь выделенный элемент в списке каталогов). См. также комментарий 5.

Недочет 2. При смене каталога на правой панели сохраняется ранее загруженное изображение.

Исправление. Добавьте новые операторы в конструктор класса `Form1` и метод `dirListBox1_Change` (листинг 13.3).

Листинг 13.3. Новые варианты конструктора формы `Form1` и метода `dirListBox1_Change`

```
public Form1()
{
  InitializeComponent();
  dirListBox1_Change(this, null);
}
private void dirListBox1_Change(object sender, EventArgs e)
{
  fileListBox1.Path = dirListBox1.Path;
```

```
  if (fileListBox1.Items.Count > 0)
    fileListBox1.SelectedIndex = 0;
  else
    pictureBox1.Image = null;
}
```

Результат. Теперь при смене каталога в списке файлов автоматически выбирается первый элемент, и его изображение выводится на правой панели. Если каталог не содержит графических файлов, то правая панель очищается. См. также комментарий 6.

> ***ПРИМЕЧАНИЕ***
>
> Поскольку при запуске программы событие `Change` компонента `dirListBox1` автоматически не возникает, приходится имитировать его возникновение, вызывая обработчик `dirListBox1_Change` в конструкторе формы (после создания и инициализации всех ее компонентов).

Недочет 3. В списке каталогов нельзя сменить каталог с помощью клавиатуры; можно лишь перемещаться по списку отображаемых каталогов.

Исправление. Определите обработчик события `KeyPress` для компонента `dirListBox1` (листинг 13.4).

Листинг 13.4. Обработчик `dirListBox1.KeyPress`

```
private void dirListBox1_KeyPress(object sender, KeyPressEventArgs e)
{
  if (e.KeyChar == '\r')
    dirListBox1.Path =
      dirListBox1.get_DirList(dirListBox1.DirListIndex);
}
```

Результат. Теперь для просмотра содержимого нового каталога достаточно выделить его в списке каталогов и нажать клавишу <Enter> (см. также комментарий 7).

Комментарии

1. В библиотеке визуальных компонентов .NET имеется специальный компонент-разделитель `Splitter`, однако с появлением в .NET 2.0 компонента `SplitContainer` необходимости в использовании отдельных компонентов-разделителей обычно не возникает.

2. Посредством вложения split-контейнеров друг в друга можно обеспечить сколь угодно сложную конфигурацию панелей с возможностью гибкой настройки их размеров. Некоторой проблемой при работе с вложенными split-контейнерами является их выделение (например, для настройки свойств в окне **Properties**). Если дочерняя панель не выделена, то выделить split-контейнер можно с помощью щелчка на его разделителе. Если же выделена дочерняя панель или находящиеся на ней компоненты, то проще несколько раз нажать клавишу <Esc>, последовательно переходя к родительским компонентам всех уровней (вплоть до формы). Приведем пример, связанный с текущим проектом. Если предположить, что в данный момент выделен компонент `pictureBox1`, то, нажимая клавишу <Esc> несколько раз, мы будем последовательно выделять следующие компоненты-контейнеры:
   - `splitContainer2.Panel2`;
   - `splitContainer2`;
   - `splitContainer1.Panel2`;
   - `splitContainer1`;
   - `Form1`.
3. Следует обратить внимание на установку для компонента `fileListBox1` свойства `IntegralHeight` равным **`False`**. Данное свойство определяет, следует ли отображать верхнюю часть последнего экранного элемента списка, если целиком этот элемент нельзя отобразить на экране. При значении **`True`** (значение по умолчанию для компонента `FileListBox`) список отображает только те элементы, которые полностью видны на экране. В нашем случае это приведет к тому, что компонент `fileListBox1` не будет выравниваться по нижней границе формы (между нижней границей компонента `fileListBox1` и нижней границей формы будет отображаться пустой промежуток). Чтобы этого не произошло, достаточно установить для свойства `IntegralHeight` значение **`False`**. Заметим, что свойство `IntegralHeight` имеется у всех компонентов-списков, в том числе и у `dirListBox1`, однако у этого компонента оно по умолчанию имеет значение **`False`** и поэтому в корректировке не нуждается.
4. Блок `try-catch` использован в обработчике `fileListBox1_SelectedIndexChanged` для того, чтобы корректно обрабатывать возможную ситуацию, когда файл с расширением, соответствующим графическим файлам, имеет неверный формат и поэтому не может быть отображен в компоненте `PictureBox`. В этой ситуации вызов конструктора класса `Bitmap` приводит к возбуждению исключения, которое сразу обрабатывается в разделе `catch` и, таким образом, не выходит за пределы исполняемого

метода. Обработка возникшего исключения состоит в том, что компонент PictureBox делается пустым, т. е. не содержащим никакого изображения.

5. Свойство FixedPanel, использованное при исправлении первого недочета, — лишь одно из свойств компонента SplitContainer, позволяющих гибко настраивать поведение связанных с ним панелей. Прежде всего, отметим свойство Orientation, которое определяет ориентацию разделителя и, соответственно, расположение дочерних панелей split-контейнера: **Vertical** (вертикальный разделитель, Panel1 находится слева, Panel2 — справа) и **Horizontal** (горизонтальный разделитель, Panel1 находится сверху, Panel2 — снизу). При описании прочих свойств мы будем использовать понятие *толщины* контейнера и панели (по аналогии с толщиной разделителя). При вертикальной ориентации разделителя под толщиной будем понимать ширину компонента, а при горизонтальной ориентации — его высоту. Всюду далее в этом комментарии под контейнером подразумевается компонент SplitContainer, а под панелями 1 и 2 — его дочерние панели, для доступа к которым у контейнера имеются свойства Panel1 и Panel2.

   Вернемся к свойству FixedPanel. Если оно равно **None** (значение по умолчанию), то при изменении толщины контейнера происходит пропорциональное изменение толщины его панелей. Если свойство FixedPanel имеет значение **Panel1** или **Panel2**, то толщина указанной панели при изменении толщины контейнера не изменяется.

   Свойства Panel1MinSize и Panel2MinSize (по умолчанию равны **25**) определяют минимальную толщину соответствующей панели. С помощью двух других парных свойств Panel1Collapsed и Panel2Collapsed можно скрывать и восстанавливать соответствующую панель, задавая свойству значение **True** или **False** (при этом оставшаяся панель займет всю область контейнера, даже если ранее она была скрыта; таким образом, скрыть сразу обе панели нельзя).

   Положение, толщина и шаг изменения разделителя определяются соответственно свойствами SplitterDistance, SplitterWidth и SplitterIncrement. По умолчанию толщина равна **4** пикселам, а шаг — **1** пикселу. Можно блокировать возможность изменять положение разделителя пользователем, положив свойство IsSplitterFixed равным **True** (при этом сохраняется возможность изменения положения разделителя программным путем).

   Как уже было отмечено, положение разделителя можно изменять не только с помощью перетаскивания его мышью, но и клавишами со стрелками, если предварительно установить на разделитель фокус с помощью клавиши <Tab>. Однако возможность принятия фокуса для разделителя не всегда является полезной. Так, при наличии в окне двух панелей с разделителем

пользователь обычно ожидает, что клавиша <Tab> позволит ему сразу перейти от одной панели к другой. В такой ситуации необходимость дважды нажимать <Tab> для смены панели будет его раздражать. Поэтому в большинстве программ разумным решением будет отключение для разделителя возможности принимать фокус; для этого достаточно положить свойство `TabStop` split-контейнера равным **`False`**.

Для отслеживания действий пользователя, связанных с изменением положения разделителя, у split-контейнера предусмотрены два свойства: `SplitterMoving`, возникающее при попытке изменить положение разделителя, и `SplitterMoved`, возникающее при успешном изменении его положения. Событие `SplitterMoving`, подобно всем событиям, возникающим перед выполнением некоторого действия, позволяет отменить его, положив в обработчике данного события значение `e.Cancel` равным `true` (`e` — второй параметр обработчика). Событие `SplitterMoved` можно только "принять к сведению", так как при его возникновении перемещение разделителя уже завершилось.

6. Компонент `FileListBox` поддерживает большинство свойств обычных компонентов-списков, в частности, свойство `SelectedIndex` — индекс выделенного элемента в списке (данное свойство доступно для чтения и записи), а также свойство-коллекцию `Items`, обеспечивающее доступ ко всем элементам списка и, в частности, позволяющее определить их количество `Count` (более подробно компоненты-списки рассматриваются в *главах 25* и *26*).

   Из специфических свойств компонента `FileListBox` следует отметить `FileName` — имя выбранного файла (без пути), `Pattern` — список масок, которым должны удовлетворять имена файлов (по умолчанию данное свойство равно **`*.*`**), а также `Path` — полный путь к каталогу, из которого берутся файлы. При изменении свойства `Path` содержимое списка автоматически обновляется. Предусмотрено также событие `PathChange`, возникающее при изменении свойства `Path`.

7. При использовании списка `DirListBox` недостаточно иметь доступ к выделенному элементу списка для получения полного пути для соответствующего каталога (поскольку в списке указывается только имя последнего подкаталога из данного пути). Проблему решает использование вспомогательного метода `get_DirList` с параметром, равным значению свойства `DirListIndex` (см. листинг 13.4). Упомянем также свойство `DirListCount`, равное количеству подкаталогов первого уровня для выбранного каталога (эти каталоги изображаются в списке под выбранным каталогом в виде закрытых папок). Для получения имен этих подкаталогов достаточно вызвать метод `get_DirList` с параметрами от `0` до `DirListCount – 1`.

13.3. Стыковка компонентов и ее особенности

В этом разделе мы разместим в нижней части формы вспомогательную панель. Поскольку вся форма заполнена дочерними панелями split-контейнеров, щелчок в любой точке формы при размещении в ней нового компонента приведет к тому, что он разместится на одной из этих дочерних панелей. Однако имеется простая возможность разместить компонент непосредственно в форме, даже если вся ее клиентская область занята другими компонентами-контейнерами: для этого достаточно при помещении компонента в форму *щелкнуть на ее заголовке*.

Выполним это действие для размещения в форме нового компонента типа `Panel`. Добавленная панель получит имя `panel1`. Настройте ее свойство `Dock`, положив его равным **`Bottom`** (для этого во вспомогательной панели окна **Properties**, связанной с этим свойством, надо щелкнуть мышью на нижнем прямоугольнике).

Ошибка. Новая панель размещается в нижней части формы (*"пристыковывается"* к нижней границе), однако располагается не рядом, а поверх остальных панелей, закрывая их нижнюю часть.

Исправление. Не снимая выделения с новой панели `panel1`, нажмите кнопку **Send to Back**  (вторая справа кнопка на панели выравнивания **Layout**).

Результат. Теперь компонент `splitContainer1`, свойство `Dock` которого имеет значение **`Fill`**, занимает всю область формы, за исключением нижней части, отведенной для панели `panel1` (рис. 13.2).

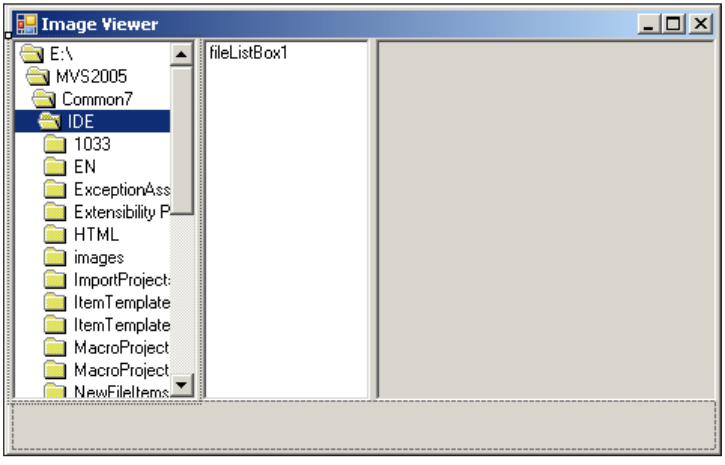


Рис. 13.2. Вид формы `Form1` для проекта IMGVIEW на промежуточном этапе разработки

Комментарий

С помощью свойства Dock, имеющегося у всех визуальных компонентов, можно легко организовать взаимное размещение компонентов в форме таким образом, чтобы они занимали всю клиентскую часть формы и не перекрывали друг друга даже при изменении размеров формы. Для этого все компоненты, кроме одного, должны быть *пристыкованы* к одной из границ формы (свойства Dock этих компонентов будут иметь значения **Top**, **Bottom**, **Left** или **Right** в зависимости от того, к какой границе формы они должны быть пристыкованы), а оставшийся компонент должен заполнять область формы, не занятую пристыкованными к краям компонентами (его свойство Dock должно быть равно **Fill**). Необходимо обсудить следующий вопрос: если два компонента будут пристыкованы к одной и той же границе, какой из них будет находиться ближе к этой границе, а какой — дальше? При визуальном проектировании формы можно дать такой ответ (правда, как мы увидим далее, он будет не совсем верным): ближе к границе будет находиться компонент, помещенный в форму раньше. Это правило соответствует интуиции разработчика: действительно, ранее добавленный элемент "захватит" свою часть формы рядом с границей и уже не отдаст ее другому претенденту, появившемуся позже. Важно подчеркнуть, что порядок присваивания значений свойствам Dock (с использованием окна **Properties**) никакой роли не играет: если, к примеру, панель panel1 была помещена в форму раньше, чем панель panel2, то даже если значение **Bottom** вначале установить для свойства panel2.Dock, а затем — для свойства panel1.Dock, ближе к нижней границе все равно будет располагаться панель panel1.

Итак, все определяется порядком размещения компонентов в форме. Но этот порядок тесным образом связан с *z*-порядком компонентов (иначе говоря, с порядком их размещения в коллекции Controls формы). Особенности *z*-порядка мы подробно рассмотрели в *разд. 10.1* (комментарий 4). В частности, мы установили, что если компонент panel1 был помещен в форму *в режиме дизайна* раньше, чем panel2, то его z-координата (а также индекс в коллекции Controls) будет иметь *большее* значение (например, если форма содержит всего два этих компонента, то panel1 будет иметь индекс 1, а panel2 — индекс 0). Но это, в свою очередь, означает, что первым в коллекцию Controls был добавлен компонент panel2 (напомним, что для добавления компонента в коллекцию Controls достаточно использовать ее метод Add). Таким образом, мы приходим еще к одному правилу, которое уже не вполне соответствует интуиции: ближе к границе будет находиться компонент, который был позже добавлен в коллекцию Controls родительского компонента (и, соответственно, имеет в ней больший индекс).

Теперь должен стать понятным способ исправления ошибки, отмеченной в настоящем разделе: для того чтобы компонент panel1 мог "захватить" нижнюю

часть формы, нам пришлось отправить его в конец *z*-последовательности, нажав кнопку **Send to Back** и установив тем самым для него наибольший индекс в коллекции `Controls`.

В заключение сделаем два замечания.

Как мы убедились, даже если компоненты добавлены в форму в порядке, который не соответствует порядку их стыковки, всегда можно исправить положение, изменив их *z*-порядок с помощью команд **Send to Back** и **Bring to Front** (именно поэтому первый, "интуитивно ясный" ответ на вопрос о порядке стыковки элементов является не вполне правильным).

Если визуальные компоненты создаются и добавляются в форму не в режиме дизайна, а программным путем (как это делается, например, в книгах [5, 6]), то для правильного порядка стыковки компонентов необходимо руководствоваться крайне "неинтуитивным", хотя и простым правилом *"работайте от центра"* (см. главу 3 в [6]): вначале надо включать в коллекцию `Controls` компонент со свойством `Dock`, равным **`Fill`**, затем — компоненты вокруг него, а в конце — компоненты, которые должны быть пристыкованы непосредственно к границе формы.

13.4. Возможность смены диска

Разместите на панели `panel1` компонент `DriveListBox` (это один из трех компонентов, добавленных нами в окно **Toolbox** в *разд. 13.1*); его положение настройте в соответствии с рис. 13.3, приведенном в следующем разделе. Новый компонент получит имя `driveListBox1`; определите для него обработчик события `SelectedIndexChanged` (листинг 13.5).

Листинг 13.5. Обработчик `driveListBox1.SelectedIndexChanged`

```
private void driveListBox1_SelectedIndexChanged(object sender,
  EventArgs e)
{
  dirListBox1.Path = driveListBox1.Drive[0] + ":\\";
}
```

Результат. Используя выпадающий список `driveListBox1`, можно выбирать любой доступный диск; при этом автоматически обновляются списки каталогов и файлов, а выбранным становится корневой каталог выбранного диска (см. также комментарий 1).

Недочет. Если выбранный дисковод (или привод для компакт-дисков) не содержит носителя, то возникает исключительная ситуация типа `IOException`

с сообщением "Device unavailable". Если после этого продолжить выполнение программы, то в выпадающем списке останется выбранным недоступный дисковод.

Исправление. Измените метод `driveListBox1_SelectedIndexChanged`, добавив в его начало новые операторы (листинг 13.6).

Листинг 13.6. Новый вариант метода `driveListBox1_SelectedIndexChanged`

```
private void driveListBox1_SelectedIndexChanged(object sender,
  EventArgs e)
{
  while (!new System.IO.DriveInfo(driveListBox1.Drive).IsReady)
    if (MessageBox.Show("Drive " + driveListBox1.Drive +
      " is not ready.", "Warning", MessageBoxButtons.RetryCancel,
      MessageBoxIcon.Warning) != DialogResult.Retry)
    {
      driveListBox1.SelectedIndexChanged -=
        driveListBox1_SelectedIndexChanged;
      driveListBox1.Drive = dirListBox1.Path[0] + ":";
      driveListBox1.SelectedIndexChanged +=
        driveListBox1_SelectedIndexChanged;
      return;
    }
  dirListBox1.Path = driveListBox1.Drive[0] + ":\\";
}
```

Результат. При попытке выбрать недоступный диск появляется окно с сообщением "Drive *<имя диска>* is not ready" и предлагаются два варианта продолжения: **Повтор** — повторить попытку чтения диска, **Отмена** — отменить выбор диска. При отмене выбора происходит возврат к ранее выбранному диску, причем выбранный каталог на этом диске не изменяется (см. комментарий 2).

ПРИМЕЧАНИЕ

При восстановлении ранее выбранного диска приходится временно отключать обработчик `driveListBox1_SelectedIndexChanged` от события `SelectedIndexChanged`. Если этого не делать, то присваивание свойству `Drive` первого символа строки `dirListBox1.Path` приведет к повторному запуску обработчика, который в этой ситуации проработает успешно и, следовательно, установит для восстановленного диска не ранее выбранный, а корневой каталог. По поводу явного отключения и подключения обработчиков *см. также разд. 4.2 и 4.3*.

Комментарии

1. Строковое свойство `Drive` компонента `driveListBox1` содержит букву выбранного диска с двоеточием, за которым может следовать название диска в квадратных скобках, например, `c: [Main]`. В методе `driveListBox1_SelectedIndexChanged` (см. листинг 13.5) используется только первый символ строки `Drive` (с индексом `0`). Если при присваивании этого свойства свойству `Path` компонента `dirListBox1` не указать обратную косую черту `\`, то среди каталогов выбранного диска будет выбран *текущий каталог* (а не корневой каталог диска). Заметим, что в методе `driveListBox1_SelectedIndexChanged` не требуется изменять свойство `Path` компонента `fileListBox1`, так как изменение свойства `dirListBox1.Path` сразу приводит к возникновению события `Change` для `dirListBox1`, в обработчике которого и будет изменено свойство `fileListBox1.Path` (см. листинг 13.3, метод `dirListBox1_Change`, первый оператор).
2. Для проверки наличия диска (см. листинг 13.6) был использован метод `IsReady` класса `DriveInfo`, определенного в пространстве имен `System.IO`. Этот метод был вызван для созданного объекта типа `DriveInfo`, соответствующего выбранному диску (по поводу класса `DriveInfo` см. также комментарий 2 в *разд. 2.2*).

13.5. Настройка режима просмотра изображений

Разместите на панели `panel1` флажок `checkBox1`, настройте его свойства (листинг 13.7, рис. 13.3) и определите для него обработчик события `CheckStateChanged` (листинг 13.8).

Листинг 13.7. Настройка свойств

```
checkBox1: Text = &Zoom Image, ThreeState = True, CheckState = Checked
```

Листинг 13.8. Обработчик `checkBox1.CheckStateChanged`

```
private void checkBox1_CheckStateChanged(object sender, EventArgs e)
{
  switch (checkBox1.CheckState)
  {
```

```
    case CheckState.Checked:
      pictureBox1.SizeMode = PictureBoxSizeMode.Zoom; break;
    case CheckState.Indeterminate:
      pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage; break;
    case CheckState.Unchecked:
      pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage; break;
  }
}
```

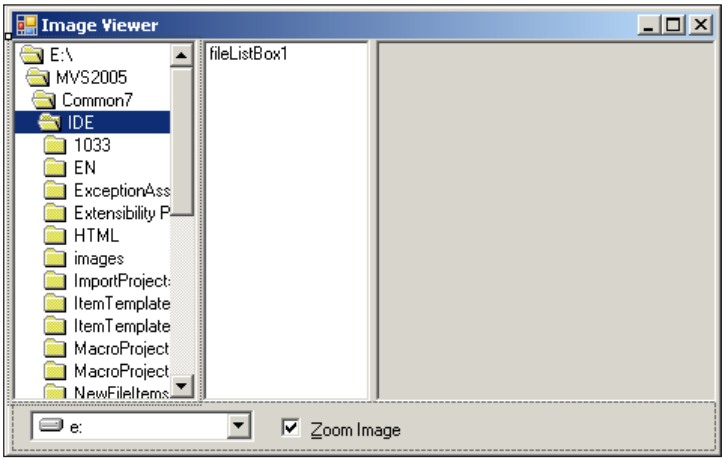


Рис. 13.3. Окончательный вид формы `Form1` для проекта IMGVIEW

Результат. Флажок **Zoom Image** может принимать три состояния, определяя режим масштабирования изображения: при установленном флажке ("галочка" на белом фоне) изображение, как и раньше, масштабируется с сохранением пропорций; при затененном флажке ("галочка" на сером фоне) изображение растягивается по всей правой панели, при этом пропорции рисунка *не сохраняются*; наконец, при сброшенном флажке изображение выводится без масштабирования и размещается в центре правой панели. Заметим, что отключать масштабирование целесообразно при просмотре значков и небольших рисунков, поскольку при их сильном увеличении изображение получается размытым.

При щелчке мышью на флажке или нажатии комбинации клавиш <Alt>+<Z> состояние флажка меняется циклически в такой последовательности: установленный — затененный — отключенный.

Комментарии

1. Для того чтобы флажок CheckBox мог принимать три состояния, необходимо положить его свойство ThreeState равным **True**. В этом случае следует анализировать не логическое свойство Checked (которое принимает значение **True** и для установленного, и для затененного состояния флажка), а свойство-перечисление CheckState с тремя возможными значениями: **Checked**, **Unchecked** и **Indeterminate** (последнее значение соответствует затененному флажку). Со свойством CheckState связано событие CheckStateChanged. Более подробно флажки рассматриваются в *главе 24*.
2. За настройку относительного расположения компонента PictureBox и загруженного в него рисунка отвечает свойство SizeMode. Варианты его значений **Zoom**, **StretchImage** и **CenterImage** были использованы в нашей программе. Имеются также варианты **Normal** (значение по умолчанию) и **AutoSize**. В обоих этих вариантах рисунок размещается в левом верхнем углу компонента PictureBox. В случае SizeMode = **Normal** рисунок обрезается, если его размер превышает размер компонента (таким образом, режимы Normal и CenterImage отличаются только способом выравнивания рисунка относительно компонента). В случае SizeMode = **AutoSize** размер *компонента* изменяется так, чтобы совпадать с размером загруженного рисунка (т. е. в этом единственном случае компонент "подстраивается" под размер рисунка, а не наоборот).

13.6. Сохранение в реестре Windows информации о состоянии программы

В начало файла Form1.cs добавьте оператор

```
using Microsoft.Win32;
```

В описание класса Form1 добавьте описание нового поля:

```
private string regKeyName = "Software\\CSExamples\\IMGVIEW";
```

Определите обработчик события FormClosed для формы Form1 (листинг 13.9).

Листинг 13.9. Обработчик Form1.FormClosed

```
private void Form1_FormClosed(object sender, FormClosedEventArgs e)
{
  RegistryKey rk = null;
  try
  {
```

```
    rk = Registry.CurrentUser.CreateSubKey(regKeyName);
    if (rk == null)
      return;
    rk.SetValue("FormWidth", Width);
    rk.SetValue("FormHeight", Height);
    rk.SetValue("Split1", splitContainer1.SplitterDistance);
    rk.SetValue("Split2", splitContainer2.SplitterDistance);
    rk.SetValue("Zoom", (int)checkBox1.CheckState);
    rk.SetValue("Path", dirListBox1.Path);
    rk.SetValue("File", fileListBox1.FileName);
  }
  finally
  {
    if (rk != null)
      rk.Close();
  }
}
```

Результат. Теперь при завершении работы программы сведения о ее текущем состоянии записываются в реестр Windows. Для того чтобы в этом убедиться, следует запустить программу regedit (Редактор реестра). Проще всего запустить данную программу, выполнив в главном меню Windows команду **Пуск | Выполнить...**, введя в появившееся окно имя программы regedit и нажав кнопку **OK** или клавишу <Enter>. В появившемся окне редактора реестра надо выбрать раздел HKEY_CURRENT_USER, а в нем подраздел Software\CSExamples\IMGVIEW. В результате на правой панели редактора реестра будут отображены поля выбранного подраздела и их значения. Подраздел должен содержать (помимо поля **По умолчанию**, которое не будет использоваться в нашей программе) семь полей: **File**, **FormHeight**, **FormWidth**, **Path**, **Split1**, **Split2**, **Zoom** (поля **File** и **Path** являются строковыми, остальные — целочисленными). В *разд. 13.7* мы добавим в программу фрагмент, позволяющий считывать из реестра эти данные.

Комментарии

1. Реестр Windows является удобным централизованным хранилищем данных, необходимых для корректной работы программ, в частности, для восстановления их настроек при очередном запуске. Если для каждого пользователя компьютера желательно хранить его собственные настройки, то их следует

размещать в разделе HKEY_CURRENT_USER. Если для всех пользователей настройки должны быть одинаковыми, то их надо размещать в разделе HKEY_LOCAL_MACHINE. В любом случае в выбранном разделе необходимо создать подраздел, связанный с конкретной программой (обычно этот подраздел помещается в подраздел Software выбранного раздела). Для доступа к данным реестра в библиотеке .NET предусмотрены классы `Registry` и `RegistryKey`, определенные в пространстве имен `Microsoft.Win32`.

Класс `Registry` позволяет выбрать один из разделов реестра первого уровня и получить связанный с ним объект типа `RegistryKey`. Для получения разделов предусмотрены статические свойства класса `Registry`, доступные только для чтения. Разделу HKEY_CURRENT_USER соответствует свойство `CurrentUser`, а разделу HKEY_LOCAL_MACHINE — свойство `LocalMachine`.

Получив с помощью класса `Registry` объект типа `RegistryKey`, можно с его помощью создавать новые подразделы и открывать существующие подразделы в режимах чтения и/или записи. Метод `CreateSubKey` открывает подраздел в режиме записи; если данный раздел не существует, то метод предварительно создает его. Метод `OpenSubKey` открывает *существующий* подраздел в режиме "только для чтения". В обоих методах в качестве строкового параметра надо указать полный путь к требуемому подразделу. Имеется также перегруженный вариант метода `OpenSubKey`, позволяющий открывать существующий подраздел в режиме "чтение и запись"; для этого в методе надо указать второй, дополнительный параметр, равный `true`. Если требуемая операция выполнена успешно, то оба метода возвращают объект типа `RegistryKey`, связанный с открытым подразделом. Если операцию выполнить не удалось, то либо возвращается значение `null` (например, в случае попытки открыть методом `OpenSubKey` несуществующий подраздел), либо возбуждается исключение (например, если программа имеет недостаточно прав для доступа к указанному подразделу в требуемом режиме). Любой успешно открытый подраздел надо закрыть с помощью метода `Close`.

Метод `SetValue` класса `RegistryKey` позволяет добавлять или изменять поля у открытого подраздела. Он имеет два параметра: имя поля и его новое значение (в качестве типа значений следует использовать `string`, `int` или массив байтов). Метод для получения значений полей будет описан в *разд. 13.7*.

2. Для отладки фрагментов программы, связанных с реестром, необходимо использовать редактор реестра regedit, поскольку он позволяет просматри-

вать и редактировать его содержимое. Если обнаружится, что подраздел или какие-либо его поля созданы с ошибками, то с помощью редактора реестра их можно легко удалить. Если при загруженном редакторе реестра тестируемая программа будет запущена повторно, то для обновления информации в окне редактора реестра достаточно нажать клавишу <F5>.

13.7. Восстановление из реестра Windows информации о состоянии программы

Измените конструктор класса `Form1` (листинг 13.10).

Листинг 13.10. Новый вариант конструктора формы `Form1`

```
public Form1()
{
  InitializeComponent();
  RegistryKey rk = null;
  string s = "";
  try
  {
    rk = Registry.CurrentUser.OpenSubKey(regKeyName);
    if (rk != null)
    {
      Width = (int)rk.GetValue("FormWidth", Width);
      Height = (int)rk.GetValue("FormHeight", Height);
      splitContainer1.SplitterDistance = (int)rk.GetValue("Split1",
        splitContainer1.SplitterDistance);
      splitContainer2.SplitterDistance = (int)rk.GetValue("Split2",
        splitContainer2.SplitterDistance);
      checkBox1.CheckState = (CheckState)rk.GetValue("Zoom",
        checkBox1.CheckState);
      s = (string)rk.GetValue("Path", "");
      if (System.IO.Directory.Exists(s))
      {
        driveListBox1.Drive = s[0] + ":";
        dirListBox1.Path = s;
```

```
      }
      s = (string)rk.GetValue("File", "");
    }
  }
  finally
  {
    if (rk != null)
      rk.Close();
  }
  fileListBox1.SelectedItem = s;
}
```

Результат. При запуске программы данные из реестра Windows используются для восстановления ее состояния. Если данные в реестре отсутствуют, то устанавливаются настройки по умолчанию. Поскольку с момента последнего сохранения данных в реестре состояние дисковой системы могло измениться, при считывании поля **Path** проверяется наличие указанного в нем каталога. Новые настройки для компонентов `driveListBox1` и `dirListBox1` устанавливаются только в случае обнаружения каталога **Path**. Проверять наличие в каталоге файла с именем, определяемым полем **File**, не требуется, так как отсутствие этого файла приведет лишь к тому, что в списке `fileListBox1` будет отсутствовать выделенный элемент.

Приведем изображение работающей программы (рис. 13.4).

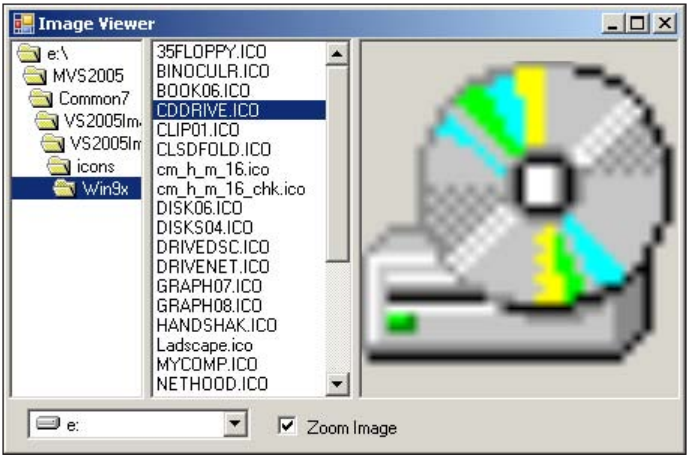
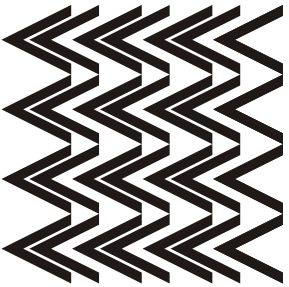


Рис. 13.4. Вид работающего приложения IMGVIEW

Комментарии

1. Для проверки существования каталога с требуемым именем используется статический метод `Exists` класса `Directory`. Одноименный статический метод класса `File` позволяет проверить существование файла.
2. Для считывания значений полей подраздела реестра предназначен метод `GetValue` класса `RegistryKey`; при этом нужный подраздел достаточно открыть только на чтение. Наиболее удобным является вариант метода `GetValue` с двумя параметрами: первый параметр содержит имя поля, а второй — значение по умолчанию (метод возвращает значение по умолчанию, если указанное поле в подразделе реестра не найдено). Вариант метода с одним параметром (именем поля) менее удобен, так как при отсутствии требуемого поля он возвращает значение `null`. Важно учитывать, что метод `GetValue` возвращает значение типа `object`, поэтому требуется явное приведение возвращаемого значения к типу соответствующего поля (`int` или `string`).
3. Редактор реестра regedit особенно полезен при отладке фрагмента программы, отвечающего за *считывание* данных из реестра. С его помощью можно удалять поля подраздела и сам подраздел, а также вносить изменения в поля, что позволяет протестировать работу программы в особых ситуациях.
4. Считывание данных из реестра обычно выполняется в начале работы программы: в конструкторе главной формы или обработчике ее события `Load`. Мы поместили соответствующий фрагмент кода в конструктор, так как при использовании обработчика события `Load` возникает одна проблема: форма при отображении на экране может не выравниваться по его центру. Это связано с тем, что центрирование созданной формы выполняется *перед* возникновением события `Load`, и поэтому если в обработчике события `Load` размеры формы изменятся, то ее центрирование будет нарушено.



Часть II

Глава 14

Работа с графическими файлами, рисование тонким пером: проект PNGEDIT1

Проект PNGEDIT1 является первым в серии проектов, посвященных работе с графической библиотекой GDI+, используемой в .NET-приложениях. В этом проекте рассматриваются основные действия с графическими файлами (создание, сохранение и загрузка) и описываются особенности основных графических классов: `Image`, `Bitmap`, `Graphics`, `PictureBox`. В проекте также реализуется отслеживание текущих координат изображения, рисование тонким пером (с использованием классов `Pen` и `Pens`) и очистка изображения.

14.1. Создание, сохранение и загрузка графических файлов

После создания проекта PNGEDIT1 добавьте в форму `Form1` три кнопки (`button1`—`button3`), панель `panel1` и невизуальные компоненты типа `OpenFileDialog` и `SaveFileDialog` (эти компоненты получат имена `openFileDialog1` и `saveFileDialog1`; они будут размещены в области невизуальных компонентов под изображением формы). На панели `panel1` разместите компонент типа `PictureBox` (он получит имя `pictureBox1`). Настройте свойства формы `Form1` и всех добавленных компонентов (листинг 14.1) и расположите компоненты в соответствии с рис. 14.1.

Добавьте к проекту новую форму (она получит имя `Form2`) и разместите в ней две метки (`label1` и `label2`), два компонента типа `NumericUpDown` (`numericUpDown1` и `numericUpDown2`) и две кнопки (`button1` и `button2`). Настройте свойства формы `Form2` и ее компонентов (листинг 14.2) и расположите компоненты в соответствии с рис. 14.2 (компонент `numericUpDown1` должен находиться рядом с меткой `label1`, а компонент `numericUpDown2` — рядом с меткой `label2`).

В начало файла Form1.cs добавьте оператор

```
using System.IO;
```

В описание класса `Form1` добавьте поле

```
private Form2 form2 = new Form2();
```

В конструктор класса `Form1` добавьте новые операторы (листинг 14.3) и определите обработчики события `Click` для кнопок `button1`, `button2` и `button3`, размещенных в форме `Form1` (листинг 14.4).

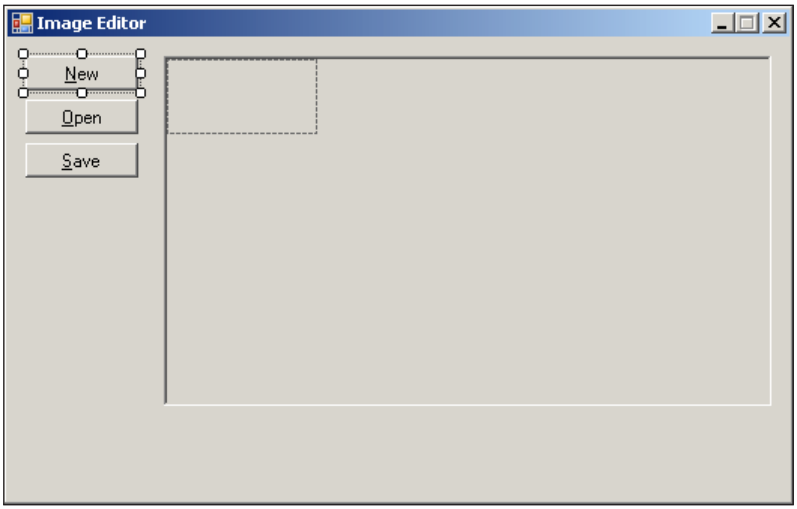


Рис. 14.1. Вид формы `Form1` для проекта PNGEDIT1 на начальном этапе разработки

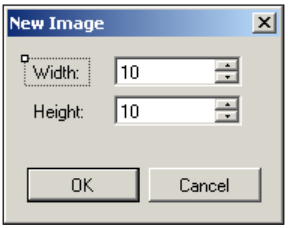


Рис. 14.2. Вид формы `Form2` для проекта PNGEDIT1

Листинг 14.1. Настройка свойств формы `Form1` и ее компонентов

```
Form1: Text = Image Editor, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
button1: Text = &New
button2: Text = &Open
button3: Text = &Save
```

```
panel1: AutoScroll = True, BorderStyle = Fixed3D
pictureBox1: Location = 0; 0, SizeMode= AutoSize
openFileDialog1: DefaultExt = png, FileName = пустая строка, Filter =
  Image files (*.bmp, *.jpg, *.png, *.gif)|*.bmp;*.jpg;*.png;*.gif
saveFileDialog1: DefaultExt = png, Filter = PNG-files (*.png)|*.png
```

Листинг 14.2. Настройка свойств формы Form2 и ее компонентов

```
Form2: Text = New Image, MaximizeBox = False, MinimizeBox = False,
  FormBorderStyle = FixedDialog, StartPosition = CenterScreen,
  ShowInTaskbar = False, AcceptButton = button1, CancelButton = button2
label1: Text = Width:
label2: Text = Height:
numericUpDown1: Minimum = 10, Maximum = 800, Increment = 10,
  Modifiers = Internal
numericUpDown2: Minimum = 10, Maximum = 600, Increment = 10,
  Modifiers = Internal
button1: Text = OK, DialogResult = OK
button2: Text = Cancel
```

Листинг 14.3. Новый вариант конструктора формы Form1

```
public Form1()
{
  InitializeComponent();
  AddOwnedForm(form2);
  openFileDialog1.InitialDirectory = saveFileDialog1.InitialDirectory =
    Directory.GetCurrentDirectory();
  form2.numericUpDown1.Value = panel1.ClientSize.Width;
  form2.numericUpDown2.Value = panel1.ClientSize.Height;
}
```

Листинг 14.4. Обработчики button1.Click, button2.Click и button3.Click (для кнопок формы Form1)

```
private void button1_Click(object sender, EventArgs e)
{
  form2.ActiveControl = form2.numericUpDown1;
```

```
  if (form2.ShowDialog() == DialogResult.OK)
  {
    saveFileDialog1.FileName = "";
    Text = "Image Editor";
    int w = (int)form2.numericUpDown1.Value,
        h = (int)form2.numericUpDown2.Value;
    Image im = new Bitmap(w, h);
    Graphics g = Graphics.FromImage(im);
    g.Clear(Color.White);
    g.Dispose();
    if (pictureBox1.Image != null)
      pictureBox1.Image.Dispose();
    pictureBox1.Image = im;
  }
}
private void button2_Click(object sender, EventArgs e)
{
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string s = openFileDialog1.FileName;
    if (pictureBox1.Image != null)
      pictureBox1.Image.Dispose();
    pictureBox1.Image = new Bitmap(s);
    Text = "Image Editor - " + s;
    saveFileDialog1.FileName = Path.ChangeExtension(s, "png");
    openFileDialog1.FileName = "";
  }
}
private void button3_Click(object sender, EventArgs e)
{
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string s = saveFileDialog1.FileName;
    pictureBox1.Image.Save(s);
    Text = "Image Editor - " + s;
  }
}
```

Результат. В редактор можно загружать графические файлы формата BMP, JPG, PNG, GIF любого размера (кнопка **Open**). Если размер файла превосходит размер выделенной для его отображения панели `panel1`, то панель снабжается полосами прокрутки, позволяющими перемещаться к любому фрагменту изображения. При загрузке файла его полное имя отображается в заголовке окна. Загруженный файл можно сохранить под новым именем в формате PNG (кнопка **Save**). В качестве каталога для загруженных и сохраняемых файлов по умолчанию предлагается рабочий каталог приложения. Кроме того, редактор позволяет создавать новые изображения (кнопка **New**). Размер нового изображения (в пикселах) запрашивается в диалоговом окне **New Image** и может изменяться от 10 × 10 до 800 × 600. При первом отображении диалогового окна предлагаются размеры, обеспечивающие заполнение всей отображаемой части панели `panel1`; в дальнейшем в окне сохраняются размеры, указанные пользователем. Созданное изображение автоматически закрашивается белым цветом. См. также комментарии 1—4.

Недочет. При запуске программы изображение в редакторе отсутствует.

Исправление. Чтобы не дублировать код, отвечающий за создание нового изображения (см. метод `button1_Click` в листинге 14.4), "делегируем" задачу создания нового изображения форме `form2`. Для этого определим обработчик события `Click` для кнопки `button1`, размещенной в форме `Form2` (листинг 14.5), и изменим метод `button1_Click` формы `Form1` (листинг 14.6; в тексте метода следует *удалить* несколько операторов, ничего не добавляя). Кроме того, в конструктор класса `Form1` добавим новый оператор:

```
form2.button1_Click(this, null);
```

Листинг 14.5. Обработчик `button1.Click` (для кнопки формы `Form2`)

```
internal void button1_Click(object sender, EventArgs e)
{
  int w = (int)numericUpDown1.Value,
    h = (int)numericUpDown2.Value;
  Image im = new Bitmap(w, h);
  Graphics g = Graphics.FromImage(im);
  g.Clear(Color.White);
  g.Dispose();
  PictureBox p = Owner.Controls["panel1"].Controls["pictureBox1"]
    as PictureBox;
  if (p.Image != null)
```

```
    p.Image.Dispose();
  p.Image = im;
}
```

Листинг 14.6. Новый вариант метода `button1_Click` формы `Form1`

```
private void button1_Click(object sender, EventArgs e)
{
  form2.ActiveControl = form2.numericUpDown1;
  if (form2.ShowDialog() == DialogResult.OK)
  {
    saveFileDialog1.FileName = "";
    Text = "Image Editor";
  }
}
```

Результат. Теперь при запуске программы в ней автоматически создается новое изображение, размеры которого совпадают с размерами клиентской части панели, содержащей данное изображение (см. также комментарий 5).

Ошибка 1. При попытке открыть файл, имеющий расширение графического файла, но содержащий данные в неверном формате (например, *пустой* файл с именем empty.bmp), возникает ошибка времени выполнения `ArgumentException` с сообщением "Parameter is not valid" (имеется в виду параметр `s` конструктора `Bitmap`). Ясно, что в подобной ситуации достаточно вывести сообщение об ошибке, однако это сообщение должно быть более информативным.

Исправление. Измените метод `button2_Click` формы `Form1` (листинг 14.7).

Листинг 14.7. Новый вариант метода `button2_Click` формы `Form1`

```
private void button2_Click(object sender, EventArgs e)
{
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string s = openFileDialog1.FileName;
    try
    {
      Image im = new Bitmap(s);
      if (pictureBox1.Image != null)
```

```
        pictureBox1.Image.Dispose();
      pictureBox1.Image = im;
    }
    catch
    {
      MessageBox.Show("File " + s + " has a wrong format.","Error");
      return;
    }
    Text = "Image Editor - " + s;
    saveFileDialog1.FileName = Path.ChangeExtension(s, "png");
    openFileDialog1.FileName = "";
  }
}
```

Результат. Теперь при попытке загрузить файл в неверном формате выводится сообщение "File <*имя файла*> has a wrong format", причем прерывание работы программы не происходит.

> ***ПРИМЕЧАНИЕ***
>
> Как обычно, в подобных ситуациях для перехвата ошибок используется механизм *явной обработки исключений (см. также главу 3)*.

Ошибка 2. При попытке сохранить загруженный файл под тем же именем (и с тем же расширением) возникает ошибка времени выполнения `ExternalException` с сообщением "A generic error occurred in GDI+". В документации .NET, посвященной методу `Save` класса `Image`, сказано, что сохранить изображение в том же файле, из которого оно было загружено, нельзя. Таким образом, выявленная ошибка связана с особенностями реализации класса `Image`.

Исправление. Измените метод `button3_Click` формы `Form1` (листинг 14.8).

Листинг 14.8. Новый вариант метода `button3_Click` формы `Form1`

```
private void button3_Click(object sender, EventArgs e)
{
  string s0 = saveFileDialog1.FileName;
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string s = saveFileDialog1.FileName;
    if (s.ToUpper() == s0.ToUpper())
```

```
    {
      s0 = Path.GetDirectoryName(s0) + "\\($$##$$).png";
      pictureBox1.Image.Save(s0);
      pictureBox1.Image.Dispose();
      File.Delete(s);
      File.Move(s0, s);
      pictureBox1.Image = new Bitmap(s);
    }
    else
      pictureBox1.Image.Save(s);
    Text = "Image Editor - " + s;
  }
}
```

Результат. Теперь при попытке сохранить изображение под тем же именем (и в том же каталоге) вначале выполняется сохранение изображения во временном файле, после чего объект `Bitmap`, связанный с данным изображением, разрушается с помощью метода `Dispose`, исходный файл с изображением удаляется с диска (методом `Delete` класса `File` из пространства имен `System.IO`), а имя временного файла заменяется на имя исходного (методом `Move` класса `File`). После этого в компонент `pictureBox1` загружается новый вариант файла. Подчеркнем, что до разрушения объекта `Bitmap` связанный с ним файл удалить с диска нельзя, так как доступ к этому файлу заблокирован. См. также комментарии 6 и 7.

Комментарии

1. В конструкторе формы `Form1` (см. листинг 14.3) для определения рабочего каталога приложения используется метод `GetCurrentDirectory` класса `Directory` из пространства имен `System.IO`. Вместо него можно было бы использовать свойство `CurrentDirectory` класса `Environment` (это было сделано в *разд. 2.2*).
2. Для отображения на экране стандартного диалогового окна, связанного с компонентом `OpenFileDialog` или `SaveFileDialog`, необходимо вызвать метод `ShowDialog` этого компонента. Если окно диалога закрывается кнопкой **Открыть** (для `OpenFileDialog`) или **Сохранить** (для `SaveFileDialog`), то метод `ShowDialog` возвращает значение `DialogResult.OK`, а имя выбранного файла сохраняется в свойстве `FileName`. Если диалог закрыт кнопкой отмены, то метод `ShowDialog` возвращает `DialogResult.Cancel`. Заметим,

что метод `ShowDialog` имеется у всех форм (см. метод `button1_Click` в листинге 14.4, а также листинг 5.3 в *разд. 5.1*).

Из часто используемых свойств компонентов `OpenFileDialog` и `SaveFileDialog` отметим `DefaultExt` (расширение файла по умолчанию; точка в начале расширения не указывается), `Filter` (*фильтры* для файлов, отображаемых в диалоговом окне; для каждого фильтра вначале указывается его описание, а затем, после вертикальной черты, — сам фильтр в виде списка масок файлов через точку с запятой; символ "вертикальная черта" также используется для отделения фильтров друг от друга), `InitialDirectory` (начальный каталог, отображаемый при открытии диалогового окна — см. листинг 14.3). Дополнительные сведения о диалоговых окнах `OpenFileDialog` и `SaveFileDialog` приводятся в *разд. 18.2—18.4*.

Обратите внимание на очистку свойства `FileName` компонента `openFileDialog1` в конце метода `button2_Click`. Это делается для того, чтобы при следующем открытии окна диалога в поле ввода не содержалось имя уже загруженного файла. Здесь же определяется имя, которое будет предложено при сохранении загруженного файла (это свойство `FileName` компонента `saveFileDialog1`). Поскольку сохранение всегда выполняется в PNG-формате, расширение в имени загруженного файла заменяется на png (для этого используется статический метод `ChangeExtension` класса `Path` из пространства имен `System.IO`).

3. Компонент `NumericUpDown`, называемый *счетчиком с прокруткой* (spin control), позволяет задавать числа из определенного диапазона, используя как явный ввод с клавиатуры, так и вспомогательные кнопки со стрелками для увеличения/уменьшения значения счетчика на фиксированную величину. Диапазон допустимых значений определяют свойства `Minimum` и `Maximum` (значения по умолчанию **`0`** и **`100`** соответственно), шаг изменения счетчика — свойство `Increment` (значение по умолчанию **`1`**), а текущее значение счетчика — свойство `Value` (значение по умолчанию **`0`**). Все эти свойства имеют тип `decimal`, что позволяет использовать счетчик для задания не только целых, но и дробных чисел. Количество знаков в дробной части числа определяется свойством `DecimalPlaces` (значение по умолчанию **`0`**, в этом случае разрешен ввод только целых чисел). Поскольку свойство `Value` имеет тип `decimal`, необходимо выполнять *явное преобразование типа* при присваивании значения этого свойства переменным других числовых типов (см., например, первый вариант метода `button1_Click` формы `Form1`, приведенный в листинге 14.4).

Имеется также компонент `DomainUpDown`, предназначенный для прокрутки текстовых строк. Этот компонент используется достаточно редко,

поскольку по удобству использования он уступает компоненту ComboBox *(см. разд. 25.1)*, имеющему аналогичные функциональные возможности.

4. Следует четко понимать роли объектов, участвующих в выводе и обработке изображения. Для визуализации любого изображения проще всего использовать компонент PictureBox, являющийся *графическим контейнером*. Само изображение входит в компонент PictureBox как свойство Image и является объектом одного из классов — потомков абстрактного класса Image. Одним из таких потомков является класс Bitmap, позволяющий хранить растровые изображения. Класс Bitmap можно создать с помощью конструктора, указав размеры изображения (это делается в первом варианте метода button1_Click формы Form1, а затем, после исправления недочета, в одноименном методе формы Form2). Кроме того, имеется вариант конструктора класса Bitmap, в котором можно указать имя существующего графического файла (отметим, что данный конструктор распознает большинство графических форматов; причем он ориентируется не на расширение, а на содержимое графического файла). Этот вариант конструктора использован в обработчике button2_Click. Созданный объект типа Bitmap достаточно присвоить свойству Image компонента PictureBox; при этом он будет сразу изображен на экране.

   Объект Bitmap можно в дальнейшем сохранить в файле с указанным именем, используя метод Save (см. обработчик button3_Click); в качестве второго, необязательного параметра можно указать графический формат, который надо применять при сохранении. Формат задается с помощью статических свойств класса ImageFormat, определенного в пространстве имен System.Drawing.Imaging. Перечислим свойства класса ImageFormat, связанные с наиболее распространенными растровыми графическими форматами: Bmp, Gif, Icon, Jpeg, Png. Если при сохранении изображения его формат не указывается (как в нашей программе), то по умолчанию используется формат Png.

   В то время как за сохранение изображения отвечает сам объект типа Bitmap, за рисование (на изображении, на форме или на компоненте) отвечает особый объект — *графическая канва* типа Graphics. Именно в классе Graphics реализованы все методы рисования (в том числе и метод Clear, использованный в нашей программе и позволяющий закрасить все изображение указанным цветом). В отличие от объектов класса Bitmap, объекты класса Graphics создаются не с помощью конструктора, а другими способами. Так, для получения объекта Graphics, связанного с растровым изображением, надо использовать статический метод FromImage класса Graphics с параметром — объектом типа Bitmap, в котором хранится требуемое изображение.

Для получения объекта `Graphics`, связанного с визуальным компонентом, можно использовать метод `CreateGraphics` этого компонента. В обработчиках события `Paint` объект `Graphics` дается в качестве одного из свойств второго параметра `e`, а именно — `e.Graphics`.

Объекты типа `Graphics` являются "короткоживущими" объектами (в отличие, например, от компонентов или объектов типа `Bitmap`). Объект `Graphics` должен быть создан непосредственно перед выполнением требуемых действий по рисованию, а после выполнения этих действий его надо разрушить с помощью метода `Dispose`. Следует заметить, что остальные объекты, связанные с графикой, также имеют метод `Dispose`, который необходимо вызывать после окончания работы с объектом. В частности, метод `Dispose` надо вызывать для прежнего изображения `Bitmap` перед подключением к компоненту `PictureBox` нового изображения. Метод `Dispose` для компонента `PictureBox` явно вызывать, как правило, не требуется: этот компонент существует на протяжении выполнения программы, а при ее завершении автоматически разрушается вместе со всеми компонентами и формами приложения.

5. Обратите внимание на способ доступа к компоненту `pictureBox1` формы `Form1` из метода `button1_Click` формы `Form2` (см. листинг 14.5): поскольку этот компонент принадлежит не форме `Form1`, а панели `panel1`, которая, в свою очередь, принадлежит форме `Form1`, для доступа к этому компоненту пришлось последовательно вызвать два свойства `Controls` (первое — для формы, второе — для ее дочернего компонента `panel1`). Для возможности вызова из формы `Form1` обработчика `button1_Click`, определенного в форме `Form2`, его модификатор доступа был изменен с `private` на `internal` (см. листинг 14.5). Благодаря наличию обработчика `button1_Click` для формы `Form2`, выполняющегося при закрытии диалогового окна **New Image** кнопкой **OK**, код обработчика `button1_Click` формы `Form1` удалось существенно упростить (см. листинг 14.6).

6. При сравнении имен файлов использована функция`s.ToUpper()`, возвращающая строку `s`, преобразованную к верхнему регистру (преобразуются как латинские, так и русские буквы). Необходимость использования подобной функции обусловлена тем, что имена файлов в системе Windows нечувствительны к регистру букв.

7. Предложенный вариант исправления ошибки 2 (см. листинг 14.8) не обеспечивает стопроцентной защиты. Если пользователь загрузит gif-файл, а затем захочет сохранить его под тем же именем *и расширением*, то условие `s.ToUpper() == s0.ToUpper()` вернет значение `false` (так как свойство `FileName` компонента `saveFileDialog1` содержит имя файла с расширени-

ем png) и будет сделана попытка сохранить изображение в том же файле, из которого оно было загружено, что, естественно, приведет к ошибке времени выполнения. Однако подобные действия пользователя маловероятны. В самом деле, диалоговое окно **Сохранить** недвусмысленно сообщает в поле **Тип файла**, что оно предназначено для сохранения данных только в PNG-формате, и в этой ситуации указывать при сохранении расширение, отличное от png, неразумно. Впрочем, защиту от подобных действий пользователя реализовать несложно: достаточно после оператора

```
string s = saveFileDialog1.FileName;
```

выполнять проверку вида `Path.GetExtension(s).ToUpper() != ".PNG"`, и, если это условие окажется истинным, выводить сообщение о том, что изображение можно сохранять только в PNG-формате, после чего немедленно выходить из обработчика.

14.2. Отслеживание текущих координат изображения

Разместите в форме `Form1` метку `label1` и настройте ее свойства (листинг 14.9) и положение (рис. 14.3; ширина метки должна быть достаточной для того, чтобы отобразить трехзначные координаты вместе с комментарием **X,Y:**).

Определите обработчик события `MouseMove` для компонента `pictureBox1` (листинг 14.10).

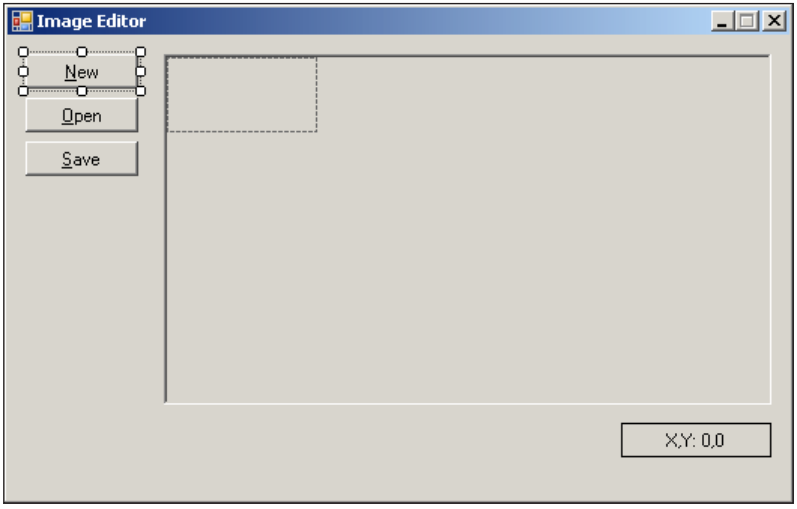


Рис. 14.3. Вид формы `Form1` для проекта PNGEDIT1 на промежуточном этапе разработки

Листинг 14.9. Настройка свойств компонента `label1` формы `Form1`

```
label1: Text = X,Y: 0,0, AutoSize = False, BorderStyle = FixedSingle,
  TextAlign = MiddleCenter
```

Листинг 14.10. Обработчик `pictureBox1.MouseMove`

```
private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{
  label1.Text = string.Format("X,Y: {0},{1}", e.X, e.Y);
}
```

Результат. При перемещении мыши над изображением (т. е. над компонентом `pictureBox1`) в метке `label1` отображаются координаты текущего пиксела изображения.

> ***ПРИМЕЧАНИЕ***
>
> Следует подчеркнуть, что выводятся именно координаты текущей точки изображения, а не позиции, отсчитываемой от левого верхнего угла компонента `panel1`. Различие между этими величинами проявляется при прокрутке изображения, имеющего размеры, которые превышают размеры панели: в подобной ситуации левый верхний угол панели будет содержать точку изображения с координатами, отличными от (0, 0).

14.3. Рисование тонким пером

Добавьте в класс `Form1` описания новых полей:

```
private Pen pen = Pens.Black;
private Point startPt;
```

Определите обработчик события `MouseDown` для компонента `pictureBox1` (листинг 14.11) и дополните метод `pictureBox1_MouseMove` (листинг 14.12).

Листинг 14.11. Обработчик `pictureBox1.MouseDown`

```
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
  startPt = e.Location;
}
```

Листинг 14.12. Новый вариант метода pictureBox1_MouseMove

```
private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{
  label1.Text = string.Format("X,Y: {0},{1}", e.X, e.Y);
  if (e.Button == MouseButtons.Left)
  {
    Graphics g = Graphics.FromImage(pictureBox1.Image);
    g.DrawLine(pen, startPt, e.Location);
    g.Dispose();
    startPt = e.Location;
    pictureBox1.Invalidate();
  }
}
```

Результат. На изображении можно рисовать линии черного цвета толщиной 1 пиксел. Для этого надо переместить курсор на начальную позицию, нажать левую кнопку мыши и, не отпуская ее, нарисовать нужную линию. Рисование возможно как на созданных, так и на загруженных изображениях. См. также комментарии 1—3.

Ошибка. Теперь при попытке загрузить некоторые bmp-файлы (например, файл Штукатурка.bmp, размещенный в системном каталоге Windows), возникает ошибка времени выполнения с сообщением "A Graphics object cannot be created from an image that has an indexed pixel format" ("Объект Graphics не может быть создан для изображения с индексированным форматом"). Подобная ошибка возникает для изображений, хранящихся в специальных *индексированных форматах* (в таких форматах цвета определяются не по их "абсолютному" RGB-значению, а по их индексу в специальной палитре цветов, включенной в файл), а также для изображений в монохромном формате GrayScale (использующем 256 оттенков серого цвета). Не обсуждая причины, по которым класс `Bitmap` ведет себя подобным образом, внесем в нашу программу добавление, позволяющее избежать в указанных ситуациях ошибки времени выполнения.

Исправление. В методе `button2_Click` формы `Form1` (см. листинг 14.7) вставьте после оператора

```
Image im = new Bitmap(s);
```

два следующих оператора:

```
Graphics g = Graphics.FromImage(im);
g.Dispose();
```

Результат. Теперь файлы, содержащие изображения, для которых невозможно создать объект `Graphics`, нельзя загрузить в программу: при попытке их загрузки выводится такое же сообщение, как и при попытке загрузить файл в неверном формате (см. описание ошибки 1 и ее исправление в *разд. 14.1*).

Недочет. Если при открытии или сохранении изображения выбрать файл в диалоговом окне, выполнив двойной щелчок на его имени, то после закрытия диалогового окна на изображении может появиться линия, соединяющая экранную точку, на которой был выполнен двойной щелчок, и точку, координаты которой хранятся в поле `startPt`. Этот недочет объясняется тем, что при подобном способе закрытия диалогового окна в компоненте `pictureBox1` может возникать событие `MouseMove`. Мы исправим этот недочет в следующей версии нашего редактора *(см. разд. 15.3)*.

Комментарии

1. В новом варианте обработчика `pictureBox1_MouseMove` (см. листинг 14.12) использован метод `DrawLine` графической канвы (т. е. объекта типа `Graphics`). Для рисования линии следует указать *перо*, которым она будет рисоваться, а также ее начальную и конечную точку. Здесь следует заметить, что в графической системе GDI+, используемой в .NET, отсутствуют понятия текущих инструментов рисования (текущего пера, кисти и т. д.). При выполнении каждой операции рисования необходимо явно указывать используемый инструмент в виде одного из параметров данной операции. Можно сказать, что в отношении инструментов рисования система GDI+ не обладает памятью. В то же время, графический редактор должен обладать памятью, поскольку это позволяет пользователю установить характеристики рисования (например, цвет линии) и применять их, пока они не будут изменены. Поэтому в класс `Form1` было добавлено поле `pen`, в котором хранится текущее перо, выбранное для рисования линий. Поле `pen` инициализируется стандартным черным пером толщины 1, которое можно получить с помощью класса `Pens`, содержащего уже созданные перья различных цветов. В дальнейшем в редакторе будут предусмотрены средства для изменения свойств текущего пера.
2. Поле `startPt` типа `Point` используется для хранения информации о предыдущей позиции мыши; значение этого поля необходимо обновлять после выполнения каждой операции рисования (см. предпоследний оператор в листинге 14.12).
3. Для того чтобы результат применения операций, связанных с рисованием, отобразился на экране, необходимо вызвать метод `Invalidate` компонента `pictureBox1` (см. последний оператор в листинге 14.12). Название

метода связано с тем, что он объявляет область, занимаемую компонентом `paintBox1` на экране, недействительной (invalid), а это вынуждает систему Windows перерисовать ее.

14.4. Очистка изображения

Разместите в форме `Form1` еще одну кнопку (`button4`), положите ее свойство `Text` равным **`&Clear`** (рис. 14.4) и определите обработчик события `Click` для этой кнопки (листинг 14.13).

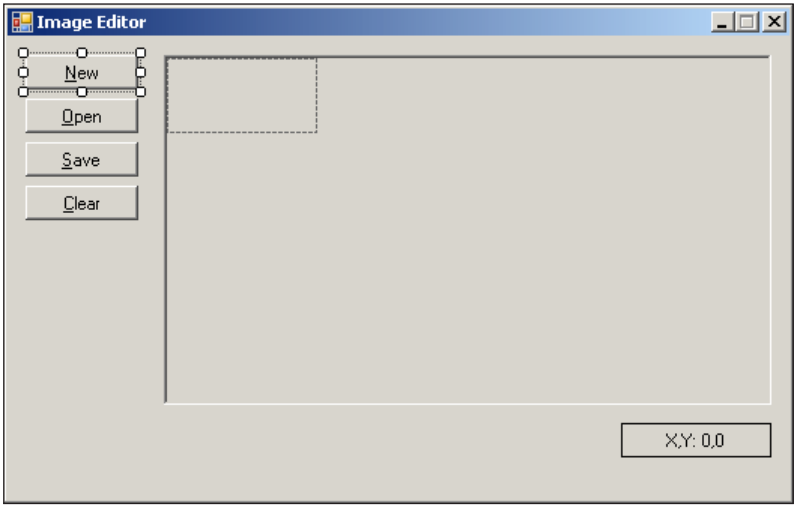


Рис. 14.4. Окончательный вид формы `Form1` для проекта PNGEDIT1

Листинг 14.13. Обработчик `button4.Click`

```
private void button4_Click(object sender, EventArgs e)
{
  using (Graphics g = Graphics.FromImage(pictureBox1.Image))
    g.Clear(Color.White);
  pictureBox1.Invalidate();
}
```

Результат. При нажатии кнопки **Clear** изображение очищается, т. е. закрашивается белым цветом.

Комментарий

В методе `button4_Click` используется полезный оператор `using`, предусмотренный в языке C# для работы с "короткоживущими" объектами (например, объектами типа `Graphics`). В заголовке этого оператора (в скобках после слова `using`) описывается и создается короткоживущая переменная, которая может использоваться только в операторе, следующем за заголовком `using` (если данную переменную требуется использовать в нескольких операторах, то их надо превратить в составной оператор, заключив в фигурные скобки `{}`). При выходе из оператора `using` для указанной переменной автоматически вызывается метод `Dispose`.

Глава 15

Цветное перо и прямые линии: проект PNGEDIT2

Проект PNGEDIT2 продолжает серию проектов, посвященных работе с библиотекой GDI+. В этом проекте реализуется рисование цветным пером произвольной толщины (классы `Pen` и `ColorDialog`) и рисование прямых линий с возможностью изображения текущего положения рисуемой линии инверсным цветом (метод `DrawReversibleLine` класса `ControlPaint`).

15.1. Рисование цветным пером

В качестве заготовки для проекта PNGEDIT2 следует использовать ранее разработанный проект PNGEDIT1 *(см. главу 14)*. Скопируйте проект PNGEDIT1 в новый каталог PNGEDIT2 и выполните действия, необходимые для переименования проекта *(см. разд. 1.1)*. Разместите в форме `Form1` две новые метки (`label2` и `label3`), компонент типа `NumericUpDown` (он получит имя `numericUpDown1`) и невизуальный компонент типа `ColorDialog` (он получит имя `colorDialog1` и будет размещен под изображением формы). Настройте свойства добавленных визуальных компонентов (листинг 15.1) и расположите их в соответствии с рис. 15.1.

Измените описание поля `pen` в классе `Form1`:

```
private Pen pen = new Pen(Color.Black);
```

Определите обработчики событий `Click` и `BackColorChanged` для метки `label2`, а также обработчик события `ValueChanged` для компонента `numericUpDown1` (листинг 15.2).

Листинг 15.1. Настройка свойств

```
label2: Text = пустая строка, AutoSize = False, Size = 40; 40,
  BackColor = Black, BorderStyle = FixedSingle
label3: Text = Width:
numericUpDown1: Maximum = 22, Minimum = 1, Value = 1, Increment = 3
```

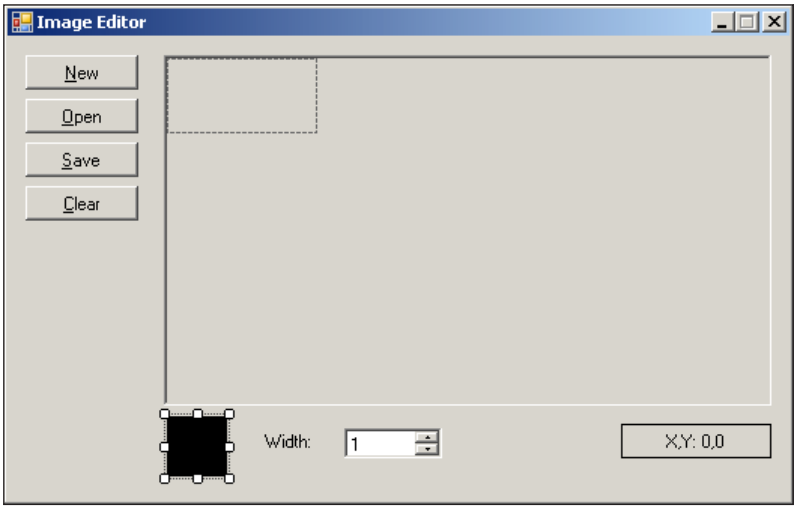


Рис. 15.1. Вид формы `Form1` для проекта PNGEDIT2 на начальном этапе разработки

Листинг 15.2. Обработчики `label2.Click`, `label2.BackColorChanged` и `numericUpDown1.ValueChanged`

```
private void label2_Click(object sender, EventArgs e)
{
  colorDialog1.Color = label2.BackColor;
  if (colorDialog1.ShowDialog() == DialogResult.OK)
    label2.BackColor = colorDialog1.Color;
}
private void label2_BackColorChanged(object sender, EventArgs e)
{
  pen.Color = label2.BackColor;
}
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
{
  pen.Width = (int)numericUpDown1.Value;
}
```

Результат. При рисовании можно выбирать:

- *цвет пера*, щелкая мышью на метке `label2`, фон которой соответствует текущему цвету пера; в результате на экране появляется диалоговое окно

Цвет, позволяющее выбрать требуемый цвет (при закрытии данного окна кнопкой **OK** фон метки `label2` закрашивается выбранным цветом);

- *толщину пера*, устанавливая ее значение с помощью компонента `numericUpDown1` (допустимыми являются значения от `1` до `22`).

Установив большую толщину и белый цвет линии, можно стирать элементы изображения. При создании или загрузке нового изображения настройки пера сохраняются. См. также комментарии 1 и 2.

Рис. 15.2. Пример линии большой толщины до исправления недочета *(слева)* и после исправления *(справа)*

Недочет. При рисовании линий большой толщины результат оказывается неудовлетворительным (рис. 15.2, *слева*).

Исправление. В начало файла Form1.cs добавьте оператор

```
using System.Drawing.Drawing2D;
```

В конструктор класса `Form1` добавьте оператор

```
pen.StartCap = pen.EndCap = LineCap.Round;
```

Результат. Теперь толстые линии рисуются надлежащим образом (рис. 15.2, *справа*). См. также комментарий 3.

Комментарии

1. Для возможности изменения свойств пера `Color` и `Width` потребовалось изменить способ его создания, так как стандартные перья, получаемые из класса `Pens`, не допускают изменения своих свойств. Для создания пера был использован один из вариантов конструктора класса `Pen`, содержащий один параметр — цвет пера (толщина по умолчанию полагается равной 1).
2. Обратите внимание на "разделение обязанностей" обработчиков для метки `label2`: обработчик `label2_Click` отвечает за выбор цвета с помощью диалогового окна и изменение фона метки в соответствии с выбранным цветом,

а обработчик label2_BackColorChanged отвечает за изменение цвета пера при изменении цвета фона метки. В дальнейшем нам может потребоваться выполнять дополнительные действия, связанные с изменением текущего цвета пера; эти действия будет достаточно указать в обработчике label2_BackColorChanged. Кроме того, цвет пера может быть изменен не только с помощью диалогового окна **Цвет**, а, например, с помощью инструмента "пипетка" *(см. разд. 17.3)*. В этом случае нам будет достаточно присвоить новый цвет свойству BackColor метки label2, и это автоматически приведет к выполнению всех действий, связанных с изменением цвета пера.

3. Отмеченный недочет был вызван неудачной формой *концов* линий. По умолчанию концы линий делаются прямоугольными (LineCap.Flat), поэтому при любом изменении направления рисования неизбежно возникают изломы. В нашей ситуации естественнее использовать закругленные концы линий; для этого достаточно после создания объекта pen установить новое значение LineCap.Round для его свойств StartCap (форма начала линии) и EndCap (форма конца линии). Перечисление LineCap определено в пространстве имен System.Drawing.Drawing2D.

15.2. Второй режим рисования: прямые линии

Разместите в форме компонент-контейнер типа GroupBox (он получит имя groupBox1) и присвойте его свойству Text значение **Mode**. В компоненте groupBox1 разместите две радиокнопки (radioButton1 и radioButton2) и присвойте их свойствам Text значения **&Pen** и **&Ruler** соответственно (рис. 15.3). Кроме того, свойство Checked радиокнопки radioButton1 положите равным **True**.

В класс Form1 добавьте три новых поля:

```
private int mode;
private Point movePt;
private Point nullPt = new Point(int.MaxValue, 0);
```

Определите в классе Form1 новый вспомогательный метод ReversibleDraw (листинг 15.3) и обработчик события CheckedChanged для радиокнопки radioButton1 (листинг 15.4), после чего свяжите созданный обработчик с событием CheckedChanged радиокнопки radioButton2.

Определите обработчик события MouseUp для компонента pictureBox1 (листинг 15.5) и измените методы pictureBox1_MouseDown и pictureBox1_MouseMove (листинг 15.6).

Наконец, в начало методов `button2_Click` (листинг 14.7) и `button3_Click` (листинг 14.8) добавьте оператор

```
startPt = nullPt;
```

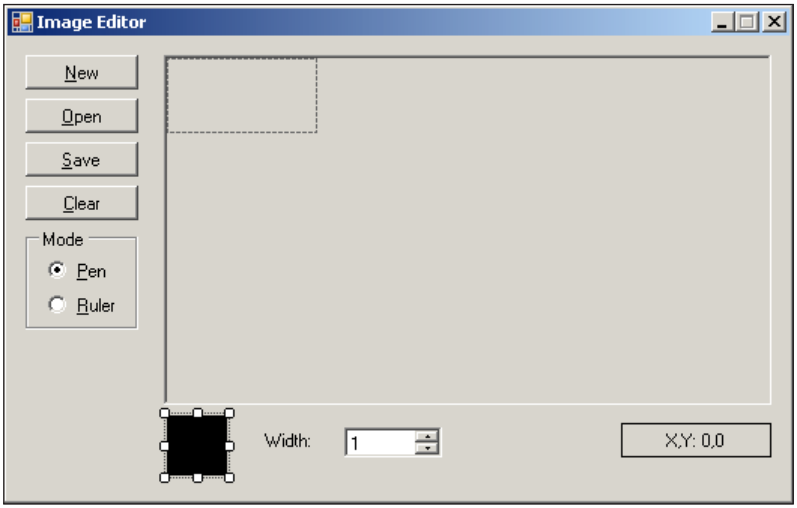


Рис. 15.3. Окончательный вид формы `Form1` для проекта PNGEDIT2

Листинг 15.3. Метод `ReversibleDraw` класса `Form1`

```
private void ReversibleDraw()
{
  Point p1 = pictureBox1.PointToScreen(startPt),
    p2= pictureBox1.PointToScreen(movePt);
  ControlPaint.DrawReversibleLine(p1, p2, Color.Black);
}
```

Листинг 15.4. Обработчик `radioButton1.CheckedChanged`

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
  RadioButton rb = sender as RadioButton;
  if (!rb.Checked)
    return;
  mode = rb.TabIndex;
}
```

Листинг 15.5. Обработчик pictureBox1.MouseUp

```
private void pictureBox1_MouseUp(object sender, MouseEventArgs e)
{
  if (startPt == nullPt)
    return;
  if (mode == 1)
  {
    Graphics g = Graphics.FromImage(pictureBox1.Image);
    g.DrawLine(pen, startPt, movePt);
    g.Dispose();
    pictureBox1.Invalidate();
  }
}
```

Листинг 15.6. Новый вариант методов pictureBox1_MouseDown и pictureBox1_MouseMove

```
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
  movePt = startPt = e.Location;
}
private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{
  label1.Text = string.Format("X,Y: {0},{1}", e.X, e.Y);
  if (startPt == nullPt)
    return;
  if (e.Button == MouseButtons.Left)
    switch (mode)
    {
      case 0:
        Graphics g = Graphics.FromImage(pictureBox1.Image);
        g.DrawLine(pen, startPt, e.Location);
        g.Dispose();
        startPt = e.Location;
        pictureBox1.Invalidate();
        break;
```

```
      case 1:
        ReversibleDraw();
        movePt = e.Location;
        ReversibleDraw();
        break;
    }
}
```

Результат. Теперь рисование можно выполнять в двух режимах: в режиме **Pen** (Перо), как раньше, рисуются линии произвольной формы, а в новом режиме **Ruler** (Линейка) рисуются прямые линии. Рисование прямых линий производится следующим образом: левая кнопка мыши нажимается в начальной точке линии и затем, при нажатой кнопке, мышь перемещается к конечной точке, где кнопка мыши отпускается. При перемещении мыши на рисунке изображается текущее положение линии (для этого используется вспомогательная линия толщины 1, цвет которой является *инверсным* по отношению к цвету под линией). Окончательно (текущим цветом и текущей толщиной) линия рисуется в момент отпускания кнопки мыши. Для рисования ломаной линии надо после отпускания кнопки мыши сразу нажать ее еще раз, и повторять этот процесс для каждого звена ломаной. Переключение между режимами рисования осуществляется радиокнопками `radioButton1` и `radioButton2`: радиокнопка, соответствующая текущему режиму, является выбранной. Для переключения между режимами рисования можно использовать клавиши-ускорители: <Alt>+<P> (перо) и <Alt>+<R> (линейка). См. также комментарии 1 и 2.

Попутно исправлен недочет, выявленный в прежнем варианте программы *(см. разд. 14.3)*. Теперь события `MouseMove` и `MouseUp`, возникающие для компонента `pictureBox1` сразу после закрытия диалогового окна `openFileDialog1` или `saveFileDialog1` (и приводящие в прежнем варианте программы к рисованию "лишней" линии), распознаются, благодаря особому значению поля `startPt`, равному `nullPt`, и в этой ситуации на изображении ничего не рисуется. См. также комментарий 3.

Недочет. Если при рисовании линии в режиме **Ruler** вывести мышь за границы компонента `pictureBox1`, то текущая линия также будет выходить за границы данного компонента (и даже за границы окна программы), причем при отпускании кнопки мыши часть этой линии, расположенная вне компонента `pictureBox1`, останется на экране. Подобное поведение объясняется тем, что использованный метод `DrawReversibleLine` обеспечивает рисование именно на экране, не привязываясь ни к каким визуальным компонентам.

Исправление. Поскольку особенности метода `DrawReversibleLine` не позволяют запретить рисование вне компонента `pictureBox1`, необходимо обеспечить, по крайней мере, удаление всех лишних линий при завершении рисования. Для этого в методе `pictureBox1_MouseUp` (см. листинг 15.5) перед оператором

```
Graphics g = Graphics.FromImage(pictureBox1.Image);
```

вставьте оператор

```
ReversibleDraw();
```

Результат. Теперь в режиме **Ruler** при отпускании кнопки мыши часть линии, выходящая за границы компонента `pictureBox1`, стирается. Правда, исключением является компонент `label1` (с информацией о текущем положении курсора мыши), на котором след от линии все же может остаться. Это связано с тем, что перерисовка компонента `label1` выполняется и при рисовании линии, поэтому вызов метода `ReversibleDraw` в методе `pictureBox1_MouseUp` не сотрет, а, наоборот, нарисует линию на метке `label1`. На данный недочет можно не обращать внимания, поскольку компонент `label1` обновляется очень часто, и уже первая его перерисовка сотрет "лишнюю" линию. Можно, однако, исправить и этот недочет: достаточно после указанного выше оператора `ReversibleDraw()` добавить еще один оператор: `label1.Invalidate()`, который заставит компонент `label1` *немедленно* перерисовать себя и тем самым стереть оставшийся след от линии.

Комментарии

1. Номер текущего режима рисования содержится в поле `mode` и определяется по свойству `TabIndex` выбранной радиокнопки. Здесь мы использовали тот факт, что при добавлении на панель `GroupBox` радиокнопок их свойства `TabIndex` принимают последовательные целочисленные значения, начиная от 0.
2. При перемещении мыши в режиме **Ruler** необходимо постоянно перерисовывать текущую линию, стирая ее прежний вариант (соответствовавший предыдущему положению курсора мыши) и рисуя новый. Для того чтобы при стирании прежнего варианта восстанавливалось исходное изображение, для линии обычно применяется *инверсный цвет*, обладающий той особенностью, что повторное рисование линии на том же месте автоматически приводит к ее удалению и восстановлению исходного изображения. В традиционной графической библиотеке Windows GDI для реализации рисования инверсным цветом было достаточно установить соответствующий режим, используя API-функцию `SetROP2` с параметром `R2_NOTXORPEN`. К сожалению, в библиотеке GDI+ аналога подобной функции нет. Поэтому

приходится довольствоваться половинчатыми средствами, предоставляемыми классом `ControlPaint`. Метод `DrawReversibleLine` этого класса позволяет рисовать в инверсном режиме *на экране*. Его первые два параметра — это концы рисуемой линии в экранных координатах, в третий параметр — наиболее вероятный цвет фона, на котором рисуется линия (в большинстве случаев в качестве этого параметра достаточно указать белый или черный цвет). Поскольку поля `startPt` и `movePt` содержат локальные координаты, связанные с компонентом `pictureBox1`, для правильного использования метода `DrawReversibleLine` выполняется переход от локальных к экранным координатам с помощью метода `PointToScreen` компонента `pictureBox1`.

3. При задании особого значения для точки `nullPt` используется поле `MaxValue`, имеющееся, наряду с полем `MinValue`, у всех числовых типов и позволяющее определить их максимальное и, соответственно, минимальное возможное значение (эти поля доступны только для чтения). Подобное присваивание гарантирует, что поле `nullPt` не будет соответствовать никакой реальной точке на изображении.

Глава 16

Прямоугольники и эллипсы, режим прозрачности: проект PNGEDIT3

Проект PNGEDIT3 продолжает серию проектов, посвященных работе с библиотекой GDI+. В этом проекте реализуется настройка фонового цвета (класс `SolidBrush`) и рисование прямоугольников и эллипсов (методами класса `Graphics`) в обычном режиме и в режиме прозрачности, т. е. без заливки внутренности фигур фоновым цветом.

16.1. Настройка фонового цвета

В качестве заготовки для проекта PNGEDIT3 следует использовать ранее разработанный проект PNGEDIT2 *(см. главу 15)*. Разместите в форме `Form1` новую метку `label4` и настройте ее свойства (листинг 16.1).

Используя кнопку **Send to Back**  (вторая справа кнопка на панели **Layout** — *см. разд. 9.1*), переместите левую верхнюю часть метки `label4` под метку `label2`, как указано на рис. 16.1.

В класс `Form1` добавьте новое поле:

```
private SolidBrush brush = new SolidBrush(Color.White);
```

Определите обработчик события `BackColorChanged` для метки `label4` (листинг 16.2) и измените методы `button4_Click` и `label2_Click` (листинг 16.3).

Кроме того, свяжите измененный обработчик `label2_Click` с событием `Click` метки `label4`.

Листинг 16.1. Настройка свойств

```
label4: Text = пустая строка, AutoSize = False, Size = 40; 40,
  BackColor = White, BorderStyle = FixedSingle
```

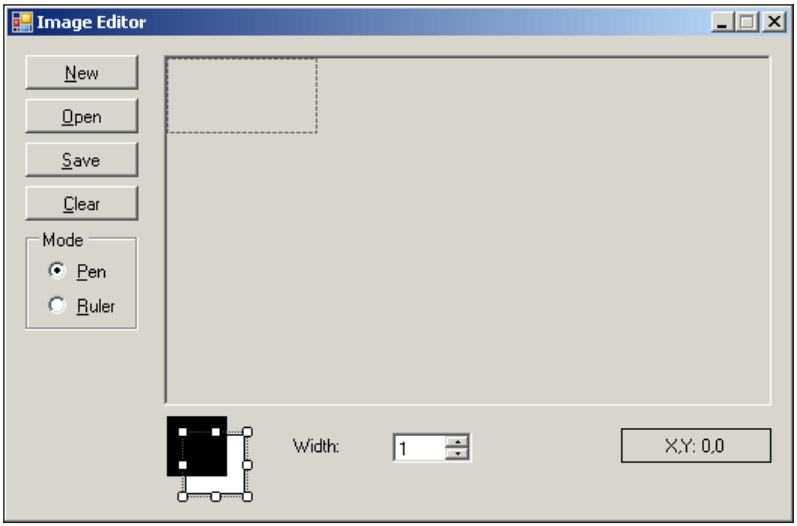


Рис. 16.1. Вид формы `Form1` для проекта PNGEDIT3 на начальном этапе разработки

Листинг 16.2. Обработчик `label4.BackColorChanged`

```
private void label4_BackColorChanged(object sender, EventArgs e)
{
  brush.Color = label4.BackColor;
}
```

Листинг 16.3. Новый вариант методов `button4_Click` и `label2_Click`

```
private void button4_Click(object sender, EventArgs e)
{
  using (Graphics g = Graphics.FromImage(pictureBox1.Image))
    g.Clear(brush.Color);
  pictureBox1.Invalidate();
}
private void label2_Click(object sender, EventArgs e)
{
  Label lb = sender as Label;
  colorDialog1.Color = lb.BackColor;
  if (colorDialog1.ShowDialog() == DialogResult.OK)
    lb.BackColor = colorDialog1.Color;
}
```

Результат. Добавленная в форму метка `label4` указывает текущий фоновый цвет, т. е. цвет инструмента рисования *кисть* (brush), используемого для заливки внутренней области замкнутых фигур. Этот цвет можно изменить с помощью диалогового окна **Цвет**, вызвав его на экран щелчком мыши на метке `label4`. Новое значение фонового цвета проявляется пока только при выполнении команды **Clear**: закрашивание рисунка теперь проводится не белым, а текущим фоновым цветом. При создании или загрузке из файла нового изображения текущее значение фонового цвета сохраняется.

Комментарии

1. Хотя в библиотеке .NET есть класс `Brush`, он является лишь абстрактным предком классов, связанных с кистями разных видов. Наиболее простым является класс `SolidBrush`, заливающий область сплошным цветом (цвет заливки определяется свойством `Color` класса `SolidBrush`). Среди других классов-кистей отметим `HatchBrush`, позволяющий заливать область одним из *штриховых* шаблонов. Класс `HatchBrush` имеет свойства `ForegroundColor` и `BackgroundColor` задающие цвета штриховых линий и промежутков между ними, а также `HatchStyle` перечислимого типа, позволяющий задать один из 53 вариантов штриховки. Интересно отметить, что, в отличие от свойства `Color` класса `SolidBrush`, доступного и для чтения, и для записи, все свойства класса `HatchBrush` доступны только для чтения (значения свойств задаются в конструкторе класса `HatchBrush` и впоследствии уже не могут изменяться). Имеются также *текстурные* и *градиентные* кисти, однако связанные с ними классы мы не будем здесь рассматривать.
2. Обратите внимание на то, что в программе используется общий обработчик события `Click` для компонентов `label2` и `label4`. Это возможно, поскольку новый вариант обработчика `label2_Click` (см. листинг 16.3) лишь отображает диалоговое окно **Цвет** и присваивает выбранный цвет той метке, которая вызвала данный обработчик (она передается в параметре `sender`), а действия, связанные с изменением цвета метки, выполняются в обработчике другого события: `BackColorChanged`.

16.2. Третий режим рисования: прямоугольники

В компоненте `groupBox1` разместите еще одну радиокнопку (`radioButton3`) и присвойте ее свойству `Text` значение **`&Figure`**. Кроме того, свяжите обработчик `radioButton1_CheckedChanged` (см. листинг 15.4) с событием `CheckedChanged` добавленной радиокнопки `radioButton3`.

В класс Form1 добавьте два новых метода: DrawFigure и PtToRect (листинг 16.4) и измените метод ReversibleDraw (листинг 16.5).

В методе pictureBox1_MouseMove (см. листинг 15.6) после строки

```
case 1:
```

добавьте строку

```
case 2:
```

и измените метод pictureBox1_MouseUp (листинг 16.6).

Листинг 16.4. Методы DrawFigure и PtToRect класса Form1

```
private void DrawFigure(Rectangle r, Graphics g)
{
  g.FillRectangle(brush, r);
  g.DrawRectangle(pen, r);
}
private Rectangle PtToRect(Point p1, Point p2)
{
  int x = Math.Min(p1.X, p2.X),
    y = Math.Min(p1.Y, p2.Y),
    w = Math.Abs(p2.X - p1.X),
    h = Math.Abs(p2.Y - p1.Y);
  return new Rectangle(x, y, w, h);
}
```

Листинг 16.5. Новый вариант метода ReversibleDraw

```
private void ReversibleDraw()
{
  Point p1 = pictureBox1.PointToScreen(startPt),
    p2= pictureBox1.PointToScreen(movePt);
  if (mode == 1)
    ControlPaint.DrawReversibleLine(p1, p2, Color.Black);
  else
    ControlPaint.DrawReversibleFrame(PtToRect(p1, p2), Color.Black,
      FrameStyle.Thick);
}
```

Листинг 16.6. Новый вариант метода pictureBox1_MouseUp

```
private void pictureBox1_MouseUp(object sender, MouseEventArgs e)
{
  if (startPt == nullPt)
    return;
  if (mode >= 1)
  {
    Graphics g = Graphics.FromImage(pictureBox1.Image);
    switch (mode)
    {
      case 1:
        g.DrawLine(pen, startPt, movePt);
        break;
      case 2:
        DrawFigure(PtToRect(startPt, movePt), g);
        break;
    }
    g.Dispose();
    pictureBox1.Invalidate();
  }
}
```

Результат. В новом режиме, который устанавливается выбором радиокнопки **Figure** (Фигура) или клавиатурной комбинацией <Alt>+<F>, можно рисовать прямоугольники. Способ рисования похож на рисование прямой линии: надо установить курсор мыши в позицию одного из углов прямоугольника, нажать левую кнопку мыши и переместить курсор в позицию противоположного по диагонали угла прямоугольника (при перемещении мыши контур прямоугольника рисуется инверсным цветом). После отпускания кнопки мыши прямоугольник окончательно рисуется текущим цветом линии, а его внутренность закрашивается текущим фоновым цветом. Ширина границы, как и толщина линии в режимах **Pen** и **Ruler**, определяется значением компонента `numericUpDown1`. См. также комментарии 1—3.

Недочет. Если толщина пера превышает 1, то полученный прямоугольник будет иметь размеры, превышающие размеры инверсного контура, рисуемого при перемещении мыши. Это связано с тем, что по умолчанию "толстый" контур замкнутых фигур рисуется по обе стороны от их границы.

Исправление. В конструктор класса `Form1` добавьте оператор

```
pen.Alignment = PenAlignment.Inset;
```

Результат. Теперь рисование контура выполняется внутри границы фигуры; таким образом, инверсный контур показывает правильные размеры фигуры независимо от текущей толщины пера (см. также комментарий 4).

Комментарии

1. Для рисования контура прямоугольника и заливки его внутренней области предназначены два различных метода класса `Graphics`: `DrawRectangle` и `FillRectangle`. Первым параметром этих методов является инструмент рисования (перо для первого метода, кисть для второго). Заметим, что аналогичным образом реализовано рисование контура и закраска других фигур: за первое действие отвечает метод, начинающийся со слова `Draw` и имеющий параметр типа `Pen`, за второе — метод, начинающийся со слова `Fill` и имеющий параметр типа `Brush`. При совместном использовании этих методов первым следует вызывать метод `Fill`, так как если вызвать его последним, то он закрасит контур, ранее нарисованный методом `Draw`. В нашей программе вызов обоих методов выполняется во вспомогательном методе `DrawFigure` (см. листинг 16.4).
2. Для создания объекта `Rectangle` по двум известным противоположным вершинам прямоугольника пришлось реализовать специальный метод `PtToRect` (см. листинг 16.4), поскольку в структуре `Rectangle` подобного метода, к сожалению, не предусмотрено. Заметим, что имеющийся в структуре `Rectangle` метод `FromLTRB` для наших целей не подходит, так как он требует задания координат левой верхней (`Left`–`Top`) и правой нижней вершины (`Right`–`Bottom`) в указанном порядке; если же задать другие вершины (или эти же вершины, но в другом порядке), то у созданного прямоугольника одно или оба измерения (`Width` и/или `Height`) будут отрицательными, а для подобных прямоугольников методы `DrawRectangle` и `FillRectangle` не выполняют никаких действий.
3. Для рисования контура в инверсном режиме используется метод `DrawReversibleFrame` класса `ControlPaint` (см. листинг 16.5). Кроме прямоугольника, определяющего рисуемый контур, и параметра типа `Color`, назначение которого совпадает с назначением аналогичного параметра метода `DrawReversibleLine` (см. комментарий 2 в *разд. 15.2*), он содержит параметр перечислимого типа `FrameStyle`, определяющий вид рисуемого контура: толстый сплошной (`Thick`) или тонкий пунктирный (`Dashed`). Пунктирный вариант контура будет использован нами в дальнейшем при

рисовании эллипсов *(см. разд. 16.3)*. Подчеркнем, что прямоугольник в методе `DrawReversibleFrame` должен задаваться в *экранных* координатах.

4. По умолчанию свойство `Alignment` пера имеет значение `PenAlignment.Center` (перечислимый тип `PenAlignment` определен в пространстве имен `System.Drawing.Drawing2D` и кроме указанных значений содержит еще варианты `Outset`, `Left` и `Right`). Интересно отметить, что при использовании всех вариантов, указанных в перечислении `PenAlignment`, кроме `PenAlignment.Inset`, результат не будет отличаться от результата для `Center`. Собственно говоря, об этом же сказано и в документации .NET (раздел `PenAlignment`): "A Pen object can be positioned to draw inside of a line or centered over the line" ("Объект Pen может быть настроен для рисования внутри линии или по ее центру").

16.3. Рисование эллипсов

Разместите в форме новую метку `label5`, очистите ее свойство `Text` и положите свойство `AutoSize` равным **`False`**, а свойство `TextAlign` равным **`MiddleCenter`**. Метка `label5` должна располагаться в левом нижнем углу формы (см. рис. 16.2, на котором метка `label5` является текущим компонентом, т. е. обрамляется белыми маркерами).

В класс `Form1` добавьте новое поле:

```
private int figureMode;
```

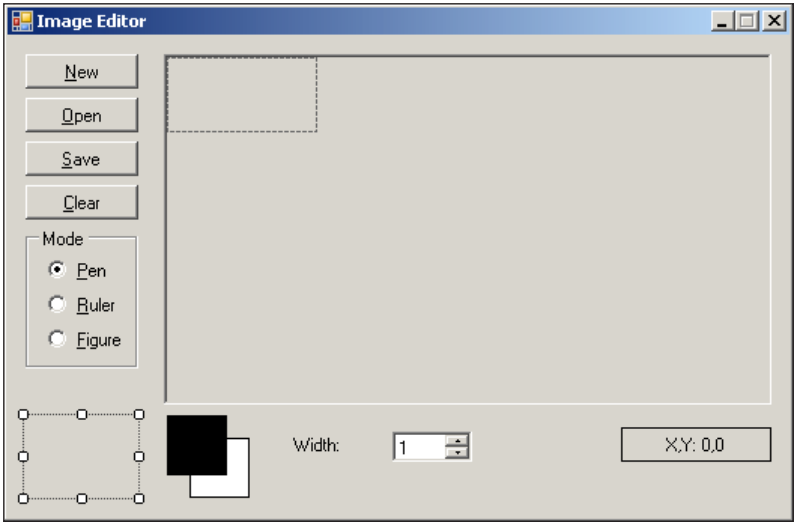


Рис. 16.2. Вид формы `Form1` для проекта PNGEDIT3 на промежуточном этапе разработки

Измените методы DrawFigure и ReversibleDraw (листинг 16.7) и определите обработчики событий Paint и MouseDown для метки label5 (листинг 16.8).

Кроме того, в методы numericUpDown1_ValueChanged, label2_BackColorChanged (см. листинг 15.2) и label4_BackColorChanged (см. листинг 16.2) добавьте оператор

```
label5.Invalidate();
```

Листинг 16.7. Новый вариант методов DrawFigure и ReversibleDraw

```
private void DrawFigure(Rectangle r, Graphics g)
{
  switch (figureMode)
  {
    case 0:
      g.FillRectangle(brush, r);
      g.DrawRectangle(pen, r);
      break;
    case 1:
      g.FillEllipse(brush, r);
      g.DrawEllipse(pen, r);
      break;
  }
}
private void ReversibleDraw()
{
  Point p1 = pictureBox1.PointToScreen(startPt),
    p2= pictureBox1.PointToScreen(movePt);
  if (mode == 1)
    ControlPaint.DrawReversibleLine(p1, p2, Color.Black);
  else
    ControlPaint.DrawReversibleFrame(PtToRect(p1, p2), Color.Black,
      (FrameStyle)((figureMode + 1) % 2));
}
```

Листинг 16.8. Обработчики label5.Paint и label5.MouseDown

```
private void label5_Paint(object sender, PaintEventArgs e)
{
  Graphics g = e.Graphics;
```

```
  Rectangle r = label5.ClientRectangle;
  DrawFigure(r, g);
}
private void label5_MouseDown(object sender, MouseEventArgs e)
{
  radioButton3.Checked = true;
  figureMode = (figureMode + 1) % 2;
  label5.Invalidate();
}
```

Результат. При щелчке мышью на метке-образце `label5` изменяется нарисованная на ней фигура (прямоугольник переходит в эллипс, эллипс — в прямоугольник) и происходит автоматический переход в режим **Figure**. В этом режиме теперь рисуется та фигура, которая изображена на метке-образце (для рисования контура эллипса и его заливки используются методы `DrawEllipse` и `FillEllipse` класса `Graphics`). Метка `label5` также отображает текущие характеристики инструментов рисования: цвет и толщину пера и цвет кисти. При рисовании эллипсов используются тонкие штриховые инверсные контуры (в отличие от более толстых сплошных инверсных контуров, отображаемых при рисовании прямоугольников). См. также комментарии 1 и 2.

Недочет. Если толщина пера равна 1, то в метке-образце `label5` не отображается правая и нижняя граница прямоугольника (а также правый и нижний фрагмент границы эллипса). Это объясняется имеющейся в GDI+ так называемой *ошибкой смещения на 1 пиксел* (см. главу 5 в [5]), которая, в частности, проявляется в том, что контуры прямоугольников и эллипсов рисуются не в прямоугольной области, указанной в качестве второго параметра методов `DrawRectangle` и `DrawEllipse`, а в области, большей указанной на 1 пиксел (по обоим измерениям).

Исправление. В методе `label5_Paint` (см. листинг 16.8) после оператора

```
Rectangle r = label5.ClientRectangle;
```

добавьте следующую строку:

```
r.Width--; r.Height--;
```

Комментарии

1. Рисование на компоненте необходимо выполнять в обработчике его события `Paint`; это гарантирует восстановление изображения при перерисовке компонента. Заметим, что перерисовка компонента автоматически выполняется в ситуации, когда компонент, ранее не видимый на экране, должен

быть на нем отображен (примером такой ситуации является перемещение ранее скрытой формы на передний план). Кроме того, перерисовку компонента можно выполнить явно, используя его метод `Invalidate`. В нашей программе метод `Invalidate` вызывается для метки-образца `label5` во всех случаях, когда происходит изменение характеристик инструментов рисования (это позволяет сразу отразить сделанное изменение на метке-образце).

С помощью второго параметра обработчика события `Paint` (типа `PaintEventArgs`) можно получить такие характеристики события, как графическая канва (`e.Graphics`) и прямоугольник, который необходимо перерисовать (`e.ClipRectangle`). Следует, однако, учитывать, что свойство `e.ClipRectangle` может возвращать только часть прямоугольной области, занимаемой компонентом, если лишь эта часть нуждается в перерисовке (например, из-за того, что данная часть была временно заслонена каким-либо диалоговым окном). Так как при изменении инструмента рисования нам необходимо перерисовывать весь компонент `label5`, в обработчике `label5_Paint` мы используем свойство `ClientRectangle`, которое всегда возвращает весь прямоугольник, занимаемый данным компонентом. Отметим также, что в обработчике `Paint` не следует вызывать для объекта `e.Graphics` его метод `Dispose`, поскольку за разрушение этого объекта отвечает метод `OnPaint` компонента (*диспетчер* события `Paint`), который создает объект `e.Graphics` и затем вызывает обработчик, передавая ему созданный объект.

2. При установке режима рисования инверсных линий, определяемого в методе `DrawReversibleFrame` третьим параметром типа `FrameStyle` (см. листинг 16.7), мы учли, что в перечислении `FrameStyle` первым элементом является `Dashed` (режим штриховых линий), а вторым — `Thick` (режим сплошных линий). Так как первый элемент перечисления в данном случае имеет номер 0, а второй — номер 1, достаточно преобразовать указанные номера к типу `FrameStyle`. Правда, штриховым линиям (элементу перечисления номер 0) в нашей программе соответствует режим рисования эллипсов (`figureMode = 1`), а сплошным линиям (элементу номер 1) — режим рисования прямоугольников (`figureMode = 0`), поэтому приходится выполнять преобразование, переводящее 0 в 1, а 1 в 0: `(figureMode + 1) % 2`.

16.4. Рисование прозрачных фигур

Разместите в форме `Form1` флажок `checkBox1` и положите его свойство `Text` равным **`Transparent:`**, а свойство `RightToLeft` равным **`Yes`**. Измените свойство `Text` метки `label5` на **`Transp.`** (см. рис. 16.3).

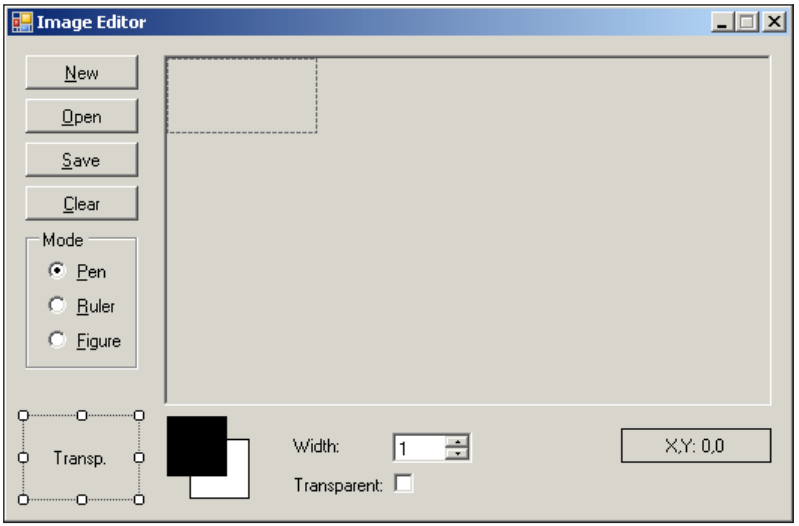


Рис. 16.3. Окончательный вид формы `Form1` для проекта PNGEDIT3

Определите обработчик события `CheckedChanged` для флажка `checkBox1` (листинг 16.9) и измените метод `DrawFigure` (листинг 16.10).

Листинг 16.9. Обработчик `checkBox1.CheckedChanged`

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
  label5.Invalidate();
}
```

Листинг 16.10. Новый вариант метода `DrawFigure`

```
private void DrawFigure(Rectangle r, Graphics g)
{
  switch (figureMode)
  {
    case 0:
      if (!checkBox1.Checked)
        g.FillRectangle(brush, r);
      g.DrawRectangle(pen, r);
      break;
    case 1:
      if (!checkBox1.Checked)
```

```
            g.FillEllipse(brush, r);
        g.DrawEllipse(pen, r);
        break;
    }
}
```

Результат. При установке флажка **Transparent** (Прозрачный) во включенное состояние внутренность рисуемых прямоугольников и эллипсов не закрашивается цветом фона. Если установлен прозрачный режим, то сквозь рисунок на метке-образце `label5` "просвечивает" ее заголовок **Transp.**. Следует заметить, что при установке прозрачного режима текущий цвет фона не изменяется; в частности, он по-прежнему используется при очистке изображения (т. е. при ее закрашивании текущим фоновым цветом) с помощью кнопки **Clear**.

ПРИМЕЧАНИЕ

Хотя в GDI+ имеется возможность устанавливать уровень прозрачности цвета *(см. разд. 9.1)*, мы использовали более простой прием: при установленном режиме прозрачности *не вызываются* методы `FillRectangle` и `FillEllipse`, обеспечивающие заливку указанной фигуры.

Приведем изображение работающей программы (рис. 16.4).

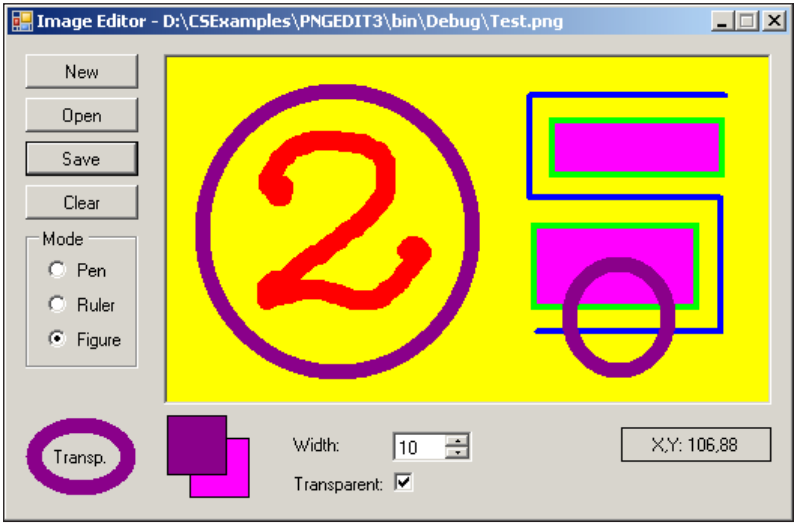


Рис. 16.4. Вид работающего приложения PNGEDIT3

Глава 17

Дополнительные графические возможности: проект PNGEDIT4

Проект PNGEDIT4 завершает серию проектов, посвященных работе с библиотекой GDI+. В этом проекте реализуются следующие дополнительные возможности: рисование квадратов и окружностей (т. е. фигур с равными измерениями), отмена последней графической операции по нажатию клавиши <Esc>, задание цвета с помощью инструмента "пипетка", добавление в рисунок текста (классы `Font` и `FontDialog`), настройка стиля изображения линии (графический выпадающий список `ComboBox`).

17.1. Рисование квадратов и окружностей

В качестве заготовки для проекта PNGEDIT4 следует использовать ранее разработанный проект PNGEDIT3 *(см. главу 16)*.

В класс `Form1` добавьте новое поле:

```
private bool equalSize;
```

Измените методы `PtToRect` и `pictureBox1_MouseMove` (листинг 17.1).

Листинг 17.1. Новый вариант методов `PtToRect` и `pictureBox1_MouseMove`

```
private Rectangle PtToRect(Point p1, Point p2)
{
  if (equalSize)
  {
    int dx = p2.X - p1.X, dy = p2.Y - p1.Y;
    if (Math.Abs(dx) > Math.Abs(dy))
      p2.X = p1.X + Math.Sign(dx) * Math.Abs(dy);
    else
      p2.Y = p1.Y + Math.Sign(dy) * Math.Abs(dx);
  }
```

```
  int x = Math.Min(p1.X, p2.X),
    y = Math.Min(p1.Y, p2.Y),
    w = Math.Abs(p2.X - p1.X),
    h = Math.Abs(p2.Y - p1.Y);
  return new Rectangle(x, y, w, h);
}
private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{
  label1.Text = string.Format("X,Y: {0},{1}", e.X, e.Y);
  if (startPt == nullPt)
    return;
  if (e.Button == MouseButtons.Left)
    switch (mode)
    {
      case 0:
        Graphics g = Graphics.FromImage(pictureBox1.Image);
        g.DrawLine(pen, startPt, e.Location);
        g.Dispose();
        startPt = e.Location;
        pictureBox1.Invalidate();
        break;
      case 1:
      case 2:
        ReversibleDraw();
        movePt = e.Location;
        equalSize = Control.ModifierKeys == Keys.Control;
        ReversibleDraw();
        break;
    }
}
```

Результат. Теперь в режиме **Figure** можно рисовать фигуры с равными измерениями (квадраты и окружности). Для этого в процессе рисования следует держать нажатой клавишу <Ctrl>. Допускается нажимать и отпускать эту клавишу после начала перемещения мыши; при этом инверсный контур будет автоматически корректироваться, изображая при нажатой клавише <Ctrl> квадратную, а в противном случае — прямоугольную рамку.

Комментарии

1. Поле `equalSize` не инициализируется при описании, так как при выполнении конструктора оно, как и все прочие поля, не инициализированные явно, автоматически получит нулевое значение соответствующего типа (т. е. в данном случае значение `false`).
2. Фрагмент, добавленный в метод `PtToRect` (см. листинг 17.1), пересчитывает одну из координат точки `p2` таким образом, чтобы точка `p2` оставалась в прежней четверти относительно точки `p1`, но при этом *модули* разностей `p2.X – p1.X` и `p2.Y – p1.Y` были равны между собой (в результате точки `p1` и `p2` будут определять вершины квадрата). Изменяется именно точка `p2`, так как точка `p1` соответствует начальной точке рисования фигуры и поэтому не может быть изменена.

   В данном фрагменте, кроме метода `Abs(X)` класса `Math`, находящего модуль числа `X`, используется метод `Sign(X)`, возвращающий `1` при положительном параметре `X`, `–1` при отрицательном и `0` при нулево.
3. Обратите внимание на то, что поле `equalSize` переопределяется в обработчике `pictureBox1_MouseMove` (см. листинг 17.1) *между* вызовами метода `ReversibleDraw`. Это обеспечивает корректное скрытие старого инверсного контура (выполняемое первым вызовом `ReversibleDraw`) перед рисованием нового.

17.2. Отмена предыдущей операции

В класс `Form1` добавьте новое поле:

```
private Bitmap oldImage;
```

В конструктор класса `Form1` добавьте оператор:

```
oldImage = new Bitmap(pictureBox1.Image);
```

В класс `Form1` добавьте новый метод `UpdateOldImage` (листинг 17.2). Вставьте вызов добавленного метода в обработчики `button1_Click`, `button2_Click`, `button4_Click` и `pictureBox1_MouseDown` (листинг 17.3).

Установите свойство `KeyPreview` формы `Form1` равным **`True`** и определите обработчик события `KeyDown` для формы `Form1` (листинг 17.4).

Листинг 17.2. Метод `UpdateOldImage` формы `Form1`

```
private void UpdateOldImage()
{
```

```
  oldImage.Dispose();
  oldImage = new Bitmap(pictureBox1.Image);
}
```

Листинг 17.3. Новый вариант методов button1_Click, button2_Click, button4_Click и pictureBox1_MouseDown

```
private void button1_Click(object sender, EventArgs e)
{
  form2.ActiveControl = form2.numericUpDown1;
  if (form2.ShowDialog() == DialogResult.OK)
  {
    saveFileDialog1.FileName = "";
    Text = "Image Editor";
    UpdateOldImage();
  }
}
private void button2_Click(object sender, EventArgs e)
{
  startPt = nullPt;
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string s = openFileDialog1.FileName;
    try
    {
      Image im = new Bitmap(s);
      Graphics g = Graphics.FromImage(im);
      g.Dispose();
      if (pictureBox1.Image != null)
        pictureBox1.Image.Dispose();
      pictureBox1.Image = im;
      UpdateOldImage();
    }
    catch
    {
      MessageBox.Show("File " + s + " has a wrong format.","Error");
      return;
    }
```

```
    Text = "Image Editor - " + s;
    saveFileDialog1.FileName = Path.ChangeExtension(s, "png");
    openFileDialog1.FileName = "";
  }
}
private void button4_Click(object sender, EventArgs e)
{
  UpdateOldImage();
  using (Graphics g = Graphics.FromImage(pictureBox1.Image))
    g.Clear(brush.Color);
  pictureBox1.Invalidate();
}
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
  movePt = startPt = e.Location;
  UpdateOldImage();
}
```

Листинг 17.4. Обработчик `Form1.KeyDown`

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
  if (e.KeyCode == Keys.Escape)
  {
    pictureBox1.Image.Dispose();
    pictureBox1.Image = new Bitmap(oldImage);
  }
}
```

Результат. Теперь любую графическую операцию (в том числе и закрашивание изображения фоновым цветом, выполняемую по нажатию кнопки **Clear**) можно отменить сразу после ее выполнения, нажав клавишу <Esc>.

Комментарий

Для возможности отмены последней операции создается вспомогательный объект-изображение `oldImage` типа `Bitmap`. При создании нового или загрузке существующего изображения в объекте `oldImage` создается его копия. Перед

началом любой операции (в момент нажатия кнопки мыши на компоненте pictureBox1, а также при нажатии кнопки **Clear**) в объект oldImage копируется прежний вариант изображения. Если после завершения операции была нажата клавиша <Esc>, то сохраненное в oldImage изображение копируется в свойство Image компонента pictureBox1, отменяя тем самым последнюю операцию.

Для получения копии изображения используется конструктор класса Bitmap с параметром — изображением, которое требуется скопировать. Присваивание без вызова конструктора (например, oldImage = pictureBox1.Image) в данном случае использовать нельзя, так как оно не создает копию изображения, а только связывает две объектные переменные с одним и тем же изображением (поскольку в переменных-объектах хранятся лишь *ссылки* на реальные объекты, размещенные в динамической памяти).

Напомним, что перед присваиванием переменной типа Image (в частности, Bitmap) нового значения необходимо вызвать ее метод Dispose для освобождения ресурсов, связанных с ее прежним изображением. Это действие следует выполнять не только для обычных переменных (как, например, oldImage), но и для *свойства* Image компонента PictureBox.

17.3. Задание цветов с помощью пипетки

Дополните метод pictureBox1_MouseDown (листинг 17.5).

Листинг 17.5. Новый вариант метода pictureBox1_MouseDown

```
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
  movePt = startPt = e.Location;
  UpdateOldImage();
  if (Control.ModifierKeys == Keys.Alt)
  {
    Color c = (pictureBox1.Image as Bitmap).GetPixel(e.X, e.Y);
    if (e.Button == MouseButtons.Left)
      label2.BackColor = c;
    else
      label4.BackColor = c;
  }
}
```

Результат. Новый цвет линии и фона теперь можно получить непосредственно из текущего изображения, "втянув его пипеткой", а именно — щелкнув соответственно левой или правой кнопкой мыши на нужном месте изображения при нажатой клавише <Alt>. Если нажата левая кнопка мыши, то можно, не отпуская ее, сразу приступить к рисованию новым цветом.

Комментарии

1. Для определения цвета в новом варианте метода `pictureBox1_MouseDown` используется метод `GetPixel(x, y)` класса `Bitmap`, возвращающий цвет пиксела изображения с координатами `(x, y)`. С помощью парного метода `SetPixel(x, y, c)` можно установить цвет `c` для пиксела изображения с координатами `(x, y)`. Обратите внимание на то, что полученный цвет достаточно присвоить свойству `BackColor` соответствующей метки (`label2` или `label4`), поскольку все остальные действия по настройке инструментов рисования будут выполнены в обработчике события `BackColorChanged` данной метки (см. комментарий 2 в *разд. 15.1*).
2. После реализации инструмента "пипетка" представляется естественным добавить к режимам графического редактора режим **Валик**, позволяющий заливать текущим цветом пера (или кисти) одноцветные области, ограниченные линиями других цветов. В графической библиотеке GDI Windows за подобное действие отвечала функция `FloodFill` и ее усовершенствованная версия `ExtFloodFill`. Имея подобную функцию, реализовать режим заливки в нашем редакторе можно буквально двумя-тремя строками кода. К сожалению, разработчики библиотеки GDI+ не сочли нужным реализовать в ней аналог функции `FloodFill`. Поскольку тема импортирования в .NET-приложение графических функций GDI является достаточно сложной и поэтому выходит за рамки данной книги, нам придется отказаться от реализации режима заливки.

17.4. Четвертый режим рисования: добавление в рисунок текста

В компоненте `groupBox1` разместите еще одну радиокнопку (`radioButton4`). Добавьте в форму `Form1` метку `label6`, поле ввода `textBox1`, кнопку `button5` и невизуальный компонент типа `FontDialog` (он получит имя `fontDialog1` и будет размещен ниже формы, в области невизуальных компонентов). Настройте свойства добавленных компонентов (листинг 17.6; рис. 17.1).

Свяжите обработчик `radioButton1_CheckedChanged` (см. листинг 15.4) с событием `CheckedChanged` добавленной радиокнопки `radioButton4`.

Добавьте в класс `Form1` новое поле:

```
private Font font;
```

В конструктор класса `Form1` добавьте оператор

```
font = Font.Clone() as Font;
```

В метод `pictureBox1_MouseDown` (листинг 17.5) добавьте новый фрагмент (листинг 17.7) и определите обработчики события `Enter` для поля ввода `textBox1` и события `Click` для кнопки `button5` (листинг 17.8).

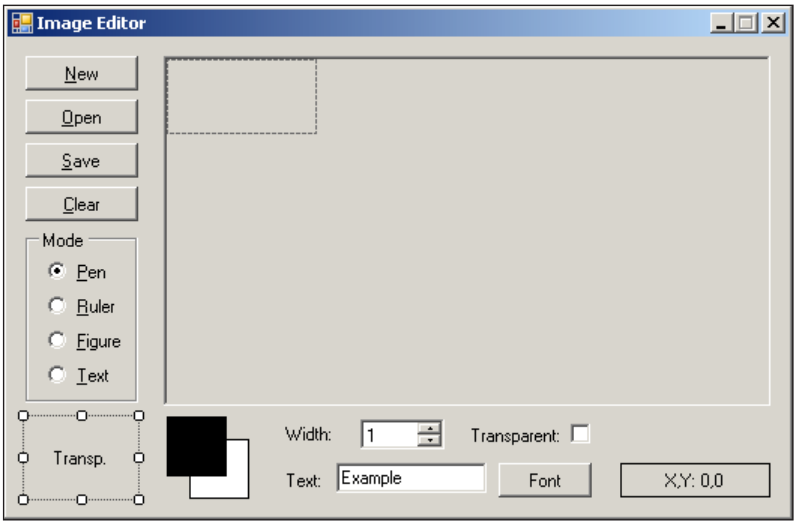


Рис. 17.1. Вид формы `Form1` для проекта PNGEDIT4

Листинг 17.6. Настройка свойств

```
radioButton4: Text = &Text
label6: Text = Text:
textBox1: Text = Example
button5: Text = Font
fontDialog1: MinSize = 8, MaxSize = 28
```

Листинг 17.7. Добавление к методу `pictureBox1_MouseDown`

```
else
  if (mode == 3)
  {
    Graphics g = Graphics.FromImage(pictureBox1.Image);
```

```
    using (SolidBrush b = new SolidBrush(pen.Color))
      g.DrawString(textBox1.Text, font, b, e.Location);
    g.Dispose();
    pictureBox1.Invalidate();
  }
```

Листинг 17.8. Обработчики `textBox1.Enter` и `button5.Click`

```
private void textBox1_Enter(object sender, EventArgs e)
{
  radioButton4.Checked = true;
}
private void button5_Click(object sender, EventArgs e)
{
  fontDialog1.Font = font;
  if (fontDialog1.ShowDialog() == DialogResult.OK)
  {
    Font f = font;
    textBox1.Font = font = fontDialog1.Font;
    f.Dispose();
  }
}
```

Результат. Новый режим **Text** позволяет вставлять в изображение текст, содержащийся в поле ввода `textBox1`. Текст вставляется при щелчке мышью на изображении; место щелчка определяет начальную позицию текста, точнее, его левый верхний угол. Текст выводится текущим цветом пера; при выводе используется шрифт, установленный для формы (Microsoft Sans Serif, 8,25 пунктов). Операция вставки текста, как и прочие графические операции, может быть отменена сразу после выполнения нажатием клавиши <Esc>. Режим **Text** автоматически устанавливается при активизации поля ввода `textBox1`, т. е. при получении им фокуса; это действие обеспечивает обработчик `textBox1_Enter` (см. листинг 17.8). См. также комментарии 1—3.

Нажатие кнопки **Font** приводит к появлению диалогового окна **Шрифт**, позволяющего выбирать нужный шрифт и настраивать такие характеристики шрифта, как начертание, размер и набор символов. Благодаря установленным значениям свойств компонента `fontDialog1` (см. листинг 17.6), размеры шрифта в диалоговом окне можно устанавливать в пределах от 8 до 28 пунктов. При открытии диалогового окна в нем отображается текущая настройка шрифта.

При выборе нового шрифта и закрытии диалогового окна кнопкой **OK** текст в поле ввода `textBox1` отображается выбранным шрифтом. См. также комментарии 4 и 5.

Комментарии

1. *Шрифт* (font) является еще одним инструментом рисования (наряду с пером и кистью). Этот инструмент реализован в виде класса `Font`. Особенностью класса `Font` является то, что все его свойства являются неизменяемыми (т. е. доступны только для чтения), и их значения можно задать только в качестве параметров конструктора при создании объекта. При создании шрифта можно указать его строковое имя (например, `"Arial"`), его размер (вещественное число, определяющее размер в пунктах, например, `10`) и начертание (типа перечисления `FontStyle` со значениями `Bold`, `Italic`, `Regular`, `Strikeout` и `Underline`, которые можно комбинировать с помощью операции `|`). Имея объект типа `Font`, можно создать его копию, используя метод `Clone` (см. оператор, добавленный в конструктор формы `Form1`).
2. Для вывода текста на экран предназначен метод `DrawString` класса `Graphics`. Кроме самого текста и шрифта, которым данный текст должен быть выведен, в этом методе указывается кисть, определяющая способ закрашивания символов текста, и координаты левого верхнего угла той области, в которой текст будет выведен. Заметим, что вместо координат угла можно указать прямоугольную область (объект типа `Rectangle`); в этом случае текст будет выводиться в пределах этой области и при необходимости может быть разбит на несколько строк. Поскольку нам требуется вывести текст цветом пера, в программе приходится создавать вспомогательную кисть соответствующего цвета (см. листинг 17.7). Оператор `using` обеспечивает автоматическое разрушение вспомогательной кисти после вывода текста (см. комментарий в *разд. 14.4*).
3. При выводе текста может оказаться полезным метод `MeasureString` класса `Graphics`. С его помощью можно определить размеры области, в которой будет выведен текст. Метод `MeasureString` возвращает структуру типа `SizeF` (подобную типу `Size`, но хранящую размеры в полях вещественного типа `float`, а не целого типа `int`); параметрами метода является выводимый текст и шрифт, которым он должен быть выведен (дополнительно могут быть указаны параметры, определяющие максимальные размеры области вывода, например максимальная ширина выводимого текста или максимальный прямоугольник, в котором должен уместиться текст).
4. С помощью диалогового окна **Шрифт** можно также выбирать цвет шрифта. Для этого следует установить свойство `ShowColor` компонента `FontDialog`

равным **True** (по умолчанию это свойство равно **False**). Поскольку свойство Font компонента FontDialog имеет тип Font и, следовательно, цветовых характеристик шрифта не содержит, для доступа к выбранному цвету в компоненте FontDialog предусмотрено особое свойство Color (значение по умолчанию — **Black**).

5. Обратите внимание на то, что в обработчике button5_Click при присваивании полю font нового объекта-шрифта связанный с ним прежний объект разрушается методом Dispose (см. листинг 17.8). Класс Font, как и другие графические классы, имеет метод Dispose, который надо вызывать при завершении работы с любым объектом данного класса для своевременного освобождения используемых им ресурсов.

17.5. Настройка стиля изображения линии

Разместите в форме Form1 над меткой label1 *выпадающий список* — компонент типа ComboBox (он получит имя comboBox1) и настройте его свойства (листинг 17.9; при определении свойства Items используется специальное диалоговое окно **String Collection Editor**, в нашем случае в это окно надо ввести указанные 5 чисел, набирая каждое число в *отдельной* строке — см. рис. 17.2).

В конструктор класса Form1 добавьте оператор

```
comboBox1.SelectedIndex = 0;
```

и определите обработчики событий DrawItem и SelectedIndexChanged для компонента comboBox1 (листинг 17.10).

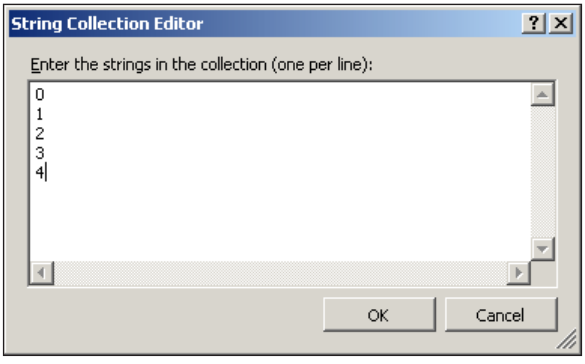


Рис. 17.2. Вид окна настройки свойства Items

Листинг 17.9. Настройка свойств

```
comboBox1: DrawMode = OwnerDrawFixed, DropDownStyle = DropDownList,
  Items: 0 1 2 3 4
```

Листинг 17.10. Обработчики comboBox1.DrawItem и comboBox1.SelectedIndexChanged

```
private void comboBox1_DrawItem(object sender, DrawItemEventArgs e)
{
  e.DrawBackground();
  // - заливка фоновым цветом области,
  // связанной с рисуемым элементом списка
  using (Pen p = new Pen(e.ForeColor, 2))
  {
    // рисование образцов линий в элементах выпадающего списка:
    p.DashStyle = (DashStyle)e.Index;
    int y = (e.Bounds.Top + e.Bounds.Bottom) / 2;
    e.Graphics.DrawLine(p, e.Bounds.Left, y, e.Bounds.Right, y);
  }
  e.DrawFocusRectangle();
  // - дополнительное выделение текущего элемента списка
}
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  pen.DashStyle = (DashStyle)comboBox1.SelectedIndex;
  label5.Invalidate();
}
```

Результат. Форма `Form1` теперь содержит выпадающий список, который позволяет выбирать все возможные стили рисования линий. Варианты стилей изображаются в выпадающем списке в графическом виде. Отметим, что в GDI+ штриховые линии могут иметь любую ширину (тогда как в прежней библиотеке GDI штриховые линии могли иметь ширину только в 1 пиксел).

> ***ПРИМЕЧАНИЕ***
>
> В режиме **Pen** линии всегда изображаются сплошными; это связано с особенностями реализации данного режима в нашей программе. Таким образом, штриховые линии можно использовать только в режимах **Ruler** и **Figure** (т. е. при рисовании прямых линий и контуров фигур).

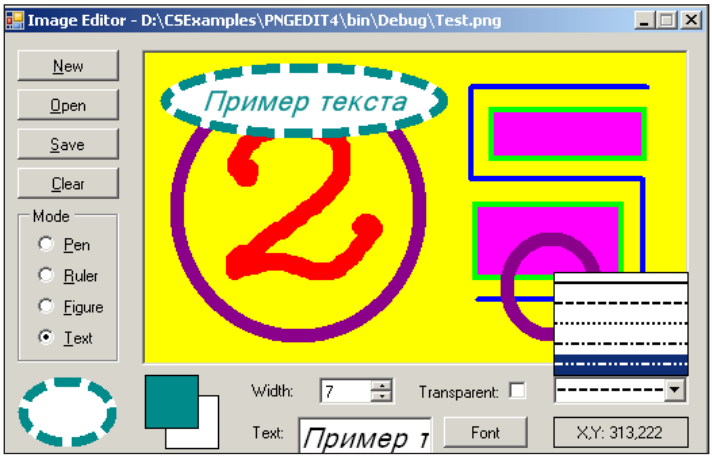


Рис. 17.3. Вид работающего приложения PNGEDIT4

Приведем изображение работающей программы с развернутым списком `comboBox1` (рис. 17.3).

Комментарии

1. Стиль линии задается в свойстве `DashStyle` класса `Pen`. Это свойство имеет перечислимый тип `DashStyle` со следующими значениями (в порядке увеличения связанных с ними номеров от 0 до 5): `Solid`, `Dash`, `Dot`, `DashDot`, `DashDotDot`, `Custom`. В нашей программе использованы все значения, кроме `Custom` (значение `Custom` позволяет применять нестандартный стиль штриховых линий, определяемый с помощью свойства `DashPattern` класса `Pen`). В обработчике `comboBox1_SelectedIndexChanged` (см. листинг 17.10) индекс выбранного элемента списка преобразуется к типу `DashStyle` (заметим, что значения элементов выпадающего списка в программе не используются; требуется лишь, чтобы количество элементов было равно 5).

   Событие `DrawItem`, обеспечивающее графическое представление элементов списка, будет более подробно рассмотрено в *разд. 26.3*.

2. Реализация различных видов штриховой *заливки фона* потребовала бы более существенной модификации нашей программы, поскольку за штриховую заливку фона отвечает кисть типа `HatchBrush`. Напомним, что в нашей программе используется кисть типа `SolidBrush`, обеспечивающая заливку фона сплошным цветом. Хотя эти классы являются потомками общего предка — класса `Brush`, они отличаются довольно сильно; достаточно отметить, что свойство `Color`, имеющееся в `SolidBrush`, отсутствует в классе `HatchBrush` (дополнительные сведения о классе `HatchBrush` приводятся в комментарии 1 в *разд. 16.1*).

Глава 18

Меню и работа с текстовыми файлами: проект TXTEDIT1

Проект TXTEDIT1 является первым в серии проектов, посвященных разработке полнофункционального текстового редактора. В этом проекте описывается создание и настройка главного меню приложения (компонент `MenuStrip`) и реализуются основные действия с текстовыми файлами (создание, сохранение, загрузка, а также вывод запроса о сохранении изменений, внесенных в текст). Рассматриваются особенности диалоговых компонентов `SaveFileDialog` и `OpenFileDialog` и многострочного варианта компонента `TextBox`.

18.1. Создание меню

После создания проекта TXTEDIT1 разместите в форме `Form1` компоненты типа `MenuStrip` и `TextBox` (добавленные компоненты получат имена `menuStrip1` и `textBox1`) и настройте свойства формы и компонента `textBox1` (листинг 18.1, первые две строки).

Следует отметить, что компоненты `MenuStrip` и `TextBox` надо помещать в форму в указанном порядке; в противном случае (если вначале разместить в форме компонент `TextBox`) верхняя часть компонента `TextBox` будет скрыта под строкой меню. Впрочем, подобную ошибку легко исправить: достаточно переместить компонент `MenuStrip` на задний план кнопкой **Send to Back**  панели **Layout** *(см. разд. 13.3)*.

Для создания меню в визуальном режиме предназначен *конструктор меню*, который активизируется при выделении компонента `menuStrip1`. В появившееся в форме поле ввода с текстом **Type Here** следует ввести имя создаваемого пункта меню. После ввода имени и нажатия клавиши <Enter> в меню будет создан новый пункт, а рядом и под ним будут выведены новые поля ввода с текстом **Type Here**, позволяющие создать очередные пункты меню первого или второго уровня (рис. 18.1). При выделении созданного пункта меню в окне **Properties** отображаются свойства данного пункта.

Каждый пункт меню `MenuStrip` является объектом типа `ToolStripMenuItem`. Имя пункта меню, в отличие от имен других компонентов, размещаемых в форме, получается не добавлением порядкового номера к имени типа (например, `label1`), а приписыванием имени команды меню слева к имени типа, например, `fileToolStripMenuItem` (это правило действует даже для пунктов меню с русскими названиями, например, `файлToolStripMenuItem`). В результате получаются очень длинные имена, поэтому сразу после создания пункта меню желательно изменить имя пункта, указав новое значение его свойства `Name`. Заметим, что свойство `Name` располагается в окне **Properties** в *начале* списка свойств; причем подпись к нему заключается в скобки: **(Name)**.

Для того чтобы сделать более наглядной связь между пунктом меню и его именем, будем использовать в примерах стандартные английские названия пунктов меню, а в качестве их имен — эти же названия, набранные с маленькой буквы и дополненные цифрой `1` (например, для пункта **File** в качестве имени будем указывать `file1`). Цифра позволяет избежать возможных конфликтов с зарезервированными словами языка; кроме того, некоторые команды мы в дальнейшем свяжем с кнопками быстрого доступа или пунктами контекстного меню, для имен которых будет удобно использовать название той же команды, но дополненное другим номером *(см. разд. 20.3 и 21.1)*.

Создайте в компоненте `menuStrip1` пункт меню первого уровня с текстом **`&File`** и с помощью окна **Properties** измените имя этого пункта (т. е. свойство `Name`) на **`file1`**. В выпадающем меню, связанном с пунктом **File**, создайте пункты с текстом **`&New`**, **`&Open`**, **`&Save`**, **`Save &As`**. Затем создайте пункт с текстом **`-`** (дефис); этот пункт будет преобразован в *горизонтальную разделительную линию*. Ниже разделительной линии создайте еще один пункт меню с текстом **`E&xit`**. Настройте свойства добавленных пунктов меню (листинг 18.1, строки с третьей по седьмую; заметим, что значения свойства `ShortcutKeys` проще и быстрее вводить непосредственно с клавиатуры, чем с помощью вспомогательной панели). Окончательный вид меню группы **File** приведен на рис. 18.1.

Находясь в конструкторе меню, можно также определять обработчики для событий, связанных с выделенным пунктом меню. Выделите пункт меню `exit1` и определите для него обработчик события `Click` (листинг 18.2).

Для выхода из конструктора меню достаточно выделить саму форму или какой-либо другой ее компонент.

Листинг 18.1. Настройка свойств

```
Form1: Text = Text Editor, StartPosition = CenterScreen
textBox1: Multiline = True, Dock = Fill
пункт меню New (группа File): Name = new1, ShortcutKeys = Ctrl+N
```

пункт меню Open (группа File): Name = **open1**, ShortcutKeys = **Ctrl+O**
пункт меню Save (группа File): Name = **save1**, ShortcutKeys = **Ctrl+S**
пункт меню Save As (группа File): Name = **saveAs1**, ShortcutKeys = **F12**
пункт меню Exit (группа File): Name = **exit1**, ShortcutKeys = **Alt+F4**

Листинг 18.2. Обработчик `exit1.Click`

```
private void exit1_Click(object sender, EventArgs e)
{
  Close();
}
```

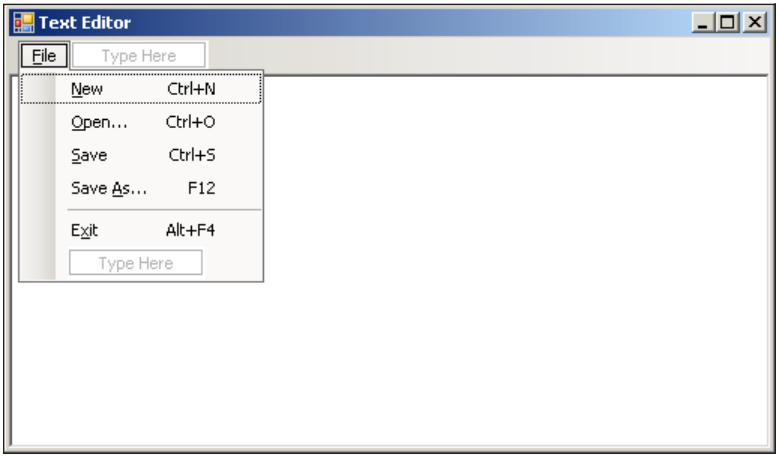


Рис. 18.1. Вид формы `Form1` для проекта TXTEDIT1

Результат. Благодаря значению **`Fill`** свойства `Dock` компонента `textBox1`, при изменении размеров окна автоматически изменяются размеры области редактирования. В программе доступно меню, команды которого можно вызывать с помощью мыши, клавиш-*ускорителей*, называемых еще клавишами быстрого доступа, или "горячими клавишами" (<Ctrl>+<N>, <F12> и т. д.), а также с помощью клавиатурных комбинаций <Alt>+<*подчеркнутая буква в названии команды*> (если развернуто меню второго уровня, то для выбора команды достаточно нажать клавишу, соответствующую подчеркнутому символу в названии команды, не нажимая клавишу <Alt>). Команда **Exit** приводит к закрытию приложения, остальные команды меню пока не выполняют никаких действий.

ПРИМЕЧАНИЕ

При запуске программы в именах пунктов меню первого уровня могут отсутствовать символы подчеркивания. В этом случае для их отображения надо нажать клавишу <Alt> и, не отпуская ее, дождаться появления символов подчеркивания, после чего нажать клавишу, соответствующую подчеркнутому символу. Заметим также, что для перехода в меню достаточно нажать либо клавишу <F10>, либо клавишу <Alt>.

Комментарии

1. Выход из программы по нажатию комбинации клавиш <Alt>+<F4> осуществляется автоматически, поэтому в явном указании клавиши-ускорителя для команды **Exit** не было необходимости. Однако подобное указание повышает наглядность программы, так как приводит к отображению клавиши-ускорителя около соответствующего пункта меню.
2. Хотя компонент `MenuStrip` является потомком класса `Control`, т. е. визуальным компонентом, в режиме дизайна он располагается не в форме, а в области под ней, предназначенной для размещения невизуальных компонентов. В то же время, меню, которое он определяет, отображается в форме. Выделить компонент `MenuStrip` можно с помощью щелчка мыши либо на его изображении в области невизуальных компонентов, либо на изображении меню в форме (вне имеющихся пунктов меню, поскольку при щелчке на пункте меню выделяется именно этот пункт, а не меню в целом). При выделении компонента `MenuStrip`, как и любого другого компонента, его свойства отображаются в окне **Properties**.
3. Компонент `MenuStrip` появился в версии 2.0 библиотеки .NET Framework. В отличие от своего предшественника — компонента `MainMenu`, являвшегося объектной оболочкой для традиционного меню окна Windows, компонент `MenuStrip` не связан с меню окна и представляет собой "обычный" визуальный компонент. Это позволяет, в частности, пристыковывать компонент `MenuStrip` к любой границе окна и более гибко настраивать внешний вид меню (например, изменяя его шрифт простой настройкой свойства `Font`, что было невозможно для компонента `MainMenu`). Пункты компонента-меню `MenuStrip` — объекты типа `ToolStripMenuItem` — имеют более богатый набор свойств по сравнению с пунктами "старого" меню `MainMenu` (в частности, они имеют свойство `Tag`, отсутствующее у пунктов меню `MainMenu`, а также свойство `Image`, позволяющее легко добавлять значок к тексту пункта меню). По этим причинам старый компонент `MainMenu` был удален из списка компонентов в окне **Toolbox** (хотя он по-прежнему включен в библиотеку классов .NET Framework и может использоваться в программах).

18.2. Сохранение текста в файле

Добавьте в форму Form1 невизуальный компонент типа SaveFileDialog (он получит имя saveFileDialog1 и будет размещен в области невизуальных компонентов под изображением формы). Настройте свойства добавленного компонента (листинг 18.3; по поводу свойств DefaultExt и Filter см. комментарий 2 в *разд. 14.1*).

В начале файла Form1.cs подключите пространство имен для файлового ввода/вывода:

```
using System.IO;
```

В конструктор класса Form1 добавьте оператор, устанавливающий рабочий каталог приложения в качестве начального каталога при сохранении файла:

```
saveFileDialog1.InitialDirectory = Directory.GetCurrentDirectory();
```

В класс Form1 добавьте новый метод SaveToFile (листинг 18.4) и определите обработчики события Click для пунктов меню saveAs1 и save1 (листинг 18.5).

Листинг 18.3. Настройка свойств

```
saveFileDialog1: DefaultExt = txt, Title = Сохранение файла,
  Filter = Text files|*.txt
```

Листинг 18.4. Метод SaveToFile класса Form1

```
private void SaveToFile(string path)
{
  StreamWriter sw = new StreamWriter(path, false, Encoding.Default);
  sw.WriteLine(textBox1.Text);
  sw.Close();
}
```

Листинг 18.5. Обработчики saveAs1.Click и save1.Click

```
private void saveAs1_Click(object sender, EventArgs e)
{
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string path = saveFileDialog1.FileName;
    SaveToFile(path);
```

```
    Text = "Text Editor - " + Path.GetFileName(path);
  }
}
private void save1_Click(object sender, EventArgs e)
{
  string path = saveFileDialog1.FileName;
  if (path == "")
    saveAs1_Click(saveAs1, null);
  else
    SaveToFile(path);
}
```

Результат. При выполнении команды **Save As...** возникает диалоговое окно **Сохранение файла** (в качестве начального каталога выбирается рабочий каталог программы). Если указать в диалоговом окне имя файла и нажать кнопку **Сохранить** или клавишу <Enter>, то набранный в редакторе текст будет сохранен в этом файле, а имя файла появится в заголовке окна программы. По умолчанию имя файла снабжается расширением txt. При выполнении команды **Save** имя файла не запрашивается, за исключением случая, когда текст еще ни разу не сохранялся.

При выходе из диалогового окна **Сохранение файла** по нажатию кнопки **Отмена** или клавиши <Esc> сохранения текста в файле не происходит. При попытке сохранить текст под именем уже существующего файла выводится запрос на подтверждение данного действия. При указании несуществующего каталога выводится предупреждающее сообщение, причем диалоговое окно не закрывается, и ошибку можно немедленно исправить.

Комментарии

1. Имя текущего файла сохраняется в свойстве `saveFileDialog1.FileName`. Использованный в программе метод `GetFileName` класса `Path` выделяет из полного имени файла собственно имя и расширение (диск и путь отбрасываются).
2. Для записи данных в текстовый файл используется текстовый поток `StreamWriter`. В его конструкторе указывается имя файла, в который будут записаны данные, режим записи (`false` — перезапись файла, `true` — дополнение прежнего содержимого новыми данными), а также формат кодирования данных при записи. Использованный в нашей программе формат `Encoding.Default` обеспечивает сохранение текста в кодировке, принятой

в Windows по умолчанию; для русской версии Windows это кодировка ANSI 1251 Cyrillic (Windows). Заметим, что если в конструкторе `StreamWriter` не указать формат кодирования, то в файл будет записан текст в кодировке UTF-8, которую могут не понимать некоторые Windows-приложения (в формате кодирования UFT-8, в отличие от однобайтового формата ANSI, для записи символов с кодовыми номерами, большими 127, используется несколько байтов; в частности, каждая русская буква кодируется двумя байтами). В то же время, именно в кодировке `Encoding.Default` сохраняется неотформатированный текст при использовании метода `SaveFile` компонента `RichTextBox`, которым мы впоследствии заменим компонент `TextBox` *(см. разд. 23.1)*.

3. Для автоматической обработки особых ситуаций, связанных с попыткой перезаписи существующего файла и с попыткой указания несуществующего каталога, необходимо, чтобы свойства `OverwritePrompt` и `CheckPathExists` компонента `saveFileDialog1` были равны **`True`** (именно таким является их значение по умолчанию).
4. Несмотря на контроль особых ситуаций, предоставляемый диалоговым окном, при сохранении файла (а также при его открытии, которое будет реализовано в следующем разделе) может возникнуть много различных ошибочных ситуаций. Для их своевременной обработки крайне желательно использовать оператор `try-catch`. В этом и последующих примерах мы не применяем оператор `try-catch` только потому, что не хотим загромождать текст программ конструкциями, правила применения которых читателю уже должны быть хорошо известны (см., в частности, *главу 3*).

18.3. Очистка области редактирования и открытие нового файла

Добавьте в форму `Form1` невизуальный компонент типа `OpenFileDialog` (он получит имя `openFileDialog1` и будет размещен в области невизуальных компонентов под изображением формы). Настройте свойства добавленного компонента (листинг 18.6).

Измените последний оператор в конструкторе класса `Form1` следующим образом:

```
openFileDialog1.InitialDirectory = saveFileDialog1.InitialDirectory =
  Directory.GetCurrentDirectory();
```

Определите обработчики события `Click` для пунктов меню `new1` и `open1` (листинг 18.7).

Листинг 18.6. Настройка свойств

```
openFileDialog1: DefaultExt = txt, FileName = пустая строка,
  Title = Открытие файла, Filter = Text files|*.txt
```

Листинг 18.7. Обработчики `new1.Click` и `open1.Click`

```
private void new1_Click(object sender, EventArgs e)
{
  textBox1.Clear();
  Text = "Text Editor";
  saveFileDialog1.FileName = "";
}
private void open1_Click(object sender, EventArgs e)
{
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string path = openFileDialog1.FileName;
    StreamReader sr = new StreamReader(path, Encoding.Default);
    textBox1.Text = sr.ReadToEnd();
    sr.Close();
    Text = "Text Editor - " + Path.GetFileName(path);
    saveFileDialog1.FileName = path;
    openFileDialog1.FileName = "";
  }
}
```

Результат. При выполнении команды **New** область редактирования очищается; при выполнении команды **Open...** появляется диалоговое окно **Открытие файла**, позволяющее выбрать файл для загрузки в редактор. В качестве начального каталога в окне **Открытие файла** устанавливается рабочий каталог приложения. При попытке открыть несуществующий файл выдается предупреждающее сообщение, однако диалоговое окно не закрывается, и ошибку можно немедленно исправить.

Заметим, что если строка из загруженного файла не умещается на экранной строке редактора, то она автоматически переносится на следующую экранную строку (разбиение строки выполняется на одном из символов пробела).

Комментарии

1. В методе `new1_Click` (см. листинг 18.7) свойство `saveFileDialog1.FileName` очищается. Это служит признаком того, что новый текст еще не сохранен в файле. Очистка свойства `openFileDialog1.FileName` в методе `open1_Click` предотвращает отображение в поле ввода имени уже загруженного файла при очередном вызове окна открытия файла.
2. Для чтения данных из текстового файла используется текстовый поток `StreamReader`. В его конструкторе указывается имя файла, из которого должны быть считаны данные, и формат кодирования данных, используемый в этом файле. При чтении данных, как и при их записи *(см. разд. 18.2)*, в нашей программе используется формат `Encoding.Default`. Метод `ReadToEnd` класса `StreamReader` считывает все содержимое файла и возвращает его в виде одной строки. Эта строка включает маркеры концов строк, что позволяет правильно разбить прочитанный текст на отдельные строки при его выводе на экран в компоненте `textBox1`.
3. Для автоматической обработки особой ситуации, связанной с попыткой чтения несуществующего файла, необходимо, чтобы свойство `CheckFileExists` компонента `openFileDialog1` было равно **`True`** (именно таким является его значение по умолчанию).
4. Задавать значение свойства `Title` для компонентов `SaveFileDialog` `OpenFileDialog` необязательно. Если это свойство не задано (т. е. является пустым), то в русской версии Windows окно сохранения файла будет иметь заголовок **Сохранить как**, а окно открытия файла — заголовок **Открыть**.
5. За автоматический перенос длинных строк отвечает свойство `WordWrap` компонента `TextBox`. Перенос выполняется, если это свойство равно **`True`** (значение по умолчанию). Если отключить автоматический перенос, положив свойство `WordWrap` равным **`False`**, то завершающая часть длинных строк может не отображаться в окне редактора. В подобной ситуации окажется полезной горизонтальная полоса прокрутки, которую можно добавить к компоненту `TextBox`, установив значение его свойства `ScrollBars` равным **`Horizontal`**. При любом значении свойства `WordWrap` полезно также добавить к компоненту `TextBox` вертикальную полосу прокрутки, позволяющую быстро перемещаться по большому тексту с помощью мыши (для добавления только вертикальной полосы свойство `ScrollBars` следует положить равным **`Vertical`**, для добавления обеих полос прокрутки — равным **`Both`**). Единственным недостатком вертикальной полосы для компонента `TextBox` является то, что она *всегда* отображается на экране (даже если загруженный текст не выходит за пределы видимой области редактирования).

18.4. Запрос о сохранении изменений, внесенных в текст

В метод `SaveToFile` класса `Form1` добавьте оператор:

```
textBox1.Modified = false;
```

В класс `Form1` добавьте новый метод `TextSaved` (листинг 18.8). Измените метод `new1_Click` (листинг 18.9) и добавьте в начало метода `open1_Click` новую строку:

```
if (TextSaved())
```

Кроме того, определите обработчик события `FormClosing` для формы `Form1` (листинг 18.10).

Листинг 18.8. Метод `TextSaved` класса `Form1`

```
private bool TextSaved()
{
  if (textBox1.Modified)
    switch (MessageBox.Show("Сохранить изменения в документе?",
      "Подтверждение", MessageBoxButtons.YesNoCancel,
      MessageBoxIcon.Question))
    {
      case DialogResult.Yes:
        save1_Click(this, null);
        return !textBox1.Modified;
      case DialogResult.Cancel:
        return false;
    }
  return true;
}
```

Листинг 18.9. Новый вариант метода `new1_Click`

```
private void new1_Click(object sender, EventArgs e)
{
  if (TextSaved())
  {
```

```
    textBox1.Clear();
    Text = "Text Editor";
    saveFileDialog1.FileName = "";
  }
}
```

Листинг 18.10. Обработчик `Form1.FormClosing`

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
  e.Cancel = !TextSaved();
}
```

Результат. Если в текущий текст были внесены изменения, то попытка очистить окно редактора командой **New**, открыть новый файл командой **Open** или выйти из программы приводит к появлению запроса о том, следует ли сохранять на диске внесенные изменения. При нажатии кнопки **Да** текущий текст сохраняется под прежним именем, а если он ни разу не был сохранен, то его имя запрашивается в диалоге **Сохранение файла**. При нажатии кнопки **Нет** изменения не сохраняются. При нажатии кнопки **Отмена** выбранное действие (команды **New**, **Open** или выход из программы) отменяется, и пользователь может продолжать редактирование текущего текста.

Комментарии

1. В методе `TextSaved` (см. листинг 18.8) используется свойство `Modified` компонента `textBox1`, принимающее значение `true`, если в текст были внесены изменения пользователем. Заметим, что любые *программные* изменения свойства `Text` компонента `textBox1` устанавливают его свойство `Modified` равным `false`. При сохранении текста в файле свойство `Modified` не изменяется, поэтому в метод `SaveToFile` был добавлен оператор, присваивающий свойству `Modified` значение `false`.
2. Функция `TextSaved` возвращает `true`, если текущий текст был ранее сохранен на диске или пользователь явно отказался от сохранения, выбрав вариант **Нет**. Если был выбран вариант **Отмена**, то функция возвращает

`false` (это означает, что пользователь желает вернуться к редактированию текущего текста). Обратите внимание на оператор

```
return !textBox1.Modified;
```

который обеспечивает корректную обработку следующей ситуации: пользователь ранее не сохранял текст в файле; при появлении запроса на сохранение текста он выбрал вариант **Да**, но вышел из появившегося диалогового окна **Сохранение файла**, нажав кнопку **Отмена**. В этом случае функция `TextSaved` вернет значение `false`, что является правильным, поскольку соответствует варианту **Отмена**, который в итоге выбрал пользователь.

Глава 19

Дополнительные возможности меню, настройка цвета и шрифта: проект TXTEDIT2

Проект TXTEDIT2 продолжает серию проектов, посвященных разработке полнофункционального текстового редактора. В этом проекте описывается создание специальных команд меню (команды-флажки и команды-переключатели) и групп меню третьего уровня. Приводится реализация команд для настройки различных характеристик шрифта и выравнивания текста в окне редактора; рассматриваются связанные с этими командами свойства классов `Font` и `TextBox`.

19.1. Установка начертания символов (команды меню — флажки)

В качестве заготовки для проекта TXTEDIT2 следует использовать ранее разработанный проект TXTEDIT1 *(см. главу 18)*. Скопируйте проект TXTEDIT1 в новый каталог TXTEDIT2 и выполните действия, необходимые для переименования проекта *(см. разд. 1.1)*.

Действуя так же, как в *разд. 18.1* при создании пункта меню **File** и связанных с ним пунктов меню второго уровня, создайте в компоненте `menuStrip1` новый пункт меню первого уровня с текстом **`F&ormat`** и с помощью окна **Properties** измените имя этого пункта (т. е. его свойство `Name`) на **`format1`**. В выпадающем меню, связанном с пунктом **Format**, создайте пункты с текстом **`&Bold`**, **`&Italic`**, **`&Underline`**. Настройте свойства добавленных пунктов меню (листинг 19.1; обратите внимание на свойство `Font`, для которого надо изменить одно из его свойств: `Bold`, `Italic` или `Underline`). Вид полученного меню приведен на рис. 19.1.

Определите обработчик события `Click` для пункта меню `bold1` (листинг 19.2), после чего свяжите созданный обработчик с событием `Click` пунктов меню `italic1` и `underline1` (связывание существующих обработчиков с событиями

пунктов меню выполняется так же, как и аналогичное связывание для обычных компонентов — *см. разд. 6.1*).

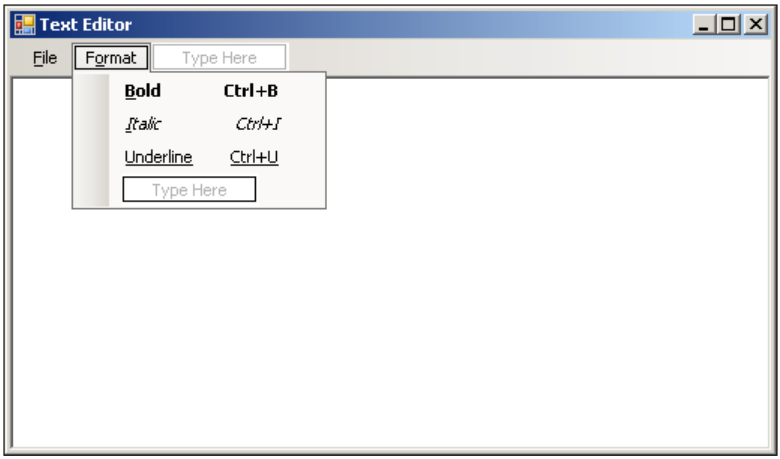


Рис. 19.1. Вид формы `Form1` для проекта TXTEDIT2 на начальном этапе разработки

Листинг 19.1. Настройка свойств

```
пункт меню Bold (группа Format): Name = bold1, ShortcutKeys = Ctrl+B,
  Font.Bold = True
пункт меню Italic (группа Format): Name = italic1,
  ShortcutKeys = Ctrl+I, Font.Italic = True
пункт меню Underline (группа Format): Name = underline1,
  ShortcutKeys = Ctrl+U, Font.Underline = True
```

Листинг 19.2. Обработчик `bold1.Click`

```
private void bold1_Click(object sender, EventArgs e)
{
  ToolStripMenuItem mi = sender as ToolStripMenuItem;
  mi.Checked = !mi.Checked;
  FontStyle fs = textBox1.Font.Style;
  fs = mi.Checked ? (fs | mi.Font.Style) : (fs & ~mi.Font.Style);
  Font f = textBox1.Font;
  textBox1.Font = new Font(f, fs);
  f.Dispose();
}
```

Результат. При выполнении добавленных в меню команд устанавливается соответствующий стиль *(начертание)* символов в редакторе: **Bold** — полужирный, **Italic** — курсивный, **Underline** — подчеркнутый. При вызове меню **Format** около команд с установленными стилями изображаются "галочки" (как для установленных флажков). При повторном выполнении команды соответствующий стиль отменяется, и галочка исчезает. Заметим, что названия указанных команд выводятся в меню тем стилем, который они настраивают (например, название команды **Bold** выводится полужирным стилем). Подобное наглядное отображение пунктов меню имеет единственный недостаток: в названии команды **Underline** нельзя определить, какая буква связана с "горячей" клавишей (поскольку подчеркнутыми являются все буквы).

Комментарии

1. У пунктов меню имеется свойство `CheckOnClick`, позволяющее при вызове команды автоматически переключать ее состояние из установленного в снятое и обратно. Для подобного поведения необходимо положить это свойство равным **`True`**. По умолчанию свойство `CheckOnClick` равно **`False`**; в этом случае при вызове команды ее состояние остается прежним, и для его изменения необходимо выполнить явные действия. Так, в методе `bold1_Click` (см. листинг 19.2) требуемые действия выполняет оператор `mi.Checked = !mi.Checked`. Мы выбрали явный вариант переключения состояния, поскольку это в дальнейшем упростит связывание команд форматирования с кнопками быстрого доступа *(см. разд. 21.2)*.
2. Для того чтобы обеспечить выполнение всех трех команд с помощью одного обработчика, мы воспользовались тем, что стиль шрифта команды меню соответствует тому стилю, который надо добавить или удалить из шрифта компонента `textBox1`. Для добавления нового стиля (т. е. элемента перечисления `FontStyle`) к имеющемуся набору стилей достаточно использовать операцию `|` (побитовое ИЛИ), для удаления стиля следует использовать операции `&` (побитовое И) и `~` (побитовое дополнение, меняющее 1 на 0, а 0 на 1).
3. Дополнительная проблема связана с тем, что ни одно свойство имеющегося шрифта нельзя изменить. Поэтому для изменения стиля шрифта приходится создавать новый шрифт с указанным стилем. Для того чтобы все прочие свойства шрифта сохранились неизменными, удобно использовать вариант конструктора класса `Font(f, s)`, специально предназначенный для изменения стиля шрифта: его первым параметром является существующий шрифт `f`, а вторым — требуемый стиль `s`. Созданный шрифт имеет все свойства шрифта `f`, за исключением стиля, который берется из второго параметра

конструктора. Напомним, что прежний шрифт после создания нового следует разрушить, вызвав для него метод `Dispose`.

19.2. Установка выравнивания текста (команды меню — переключатели)

Дополните выпадающее меню, связанное с пунктом **Format** *(см. разд. 19.1)*. Вначале добавьте в него пункт с текстом **-** (дефис); этот пункт будет преобразован в горизонтальную разделительную линию. Затем добавьте пункты меню с текстом **`&Left justify`**, **`C&enter`** и **`&Right justify`** и настройте свойства этих пунктов (листинг 19.3).

В класс `Form1` добавьте новое поле:

```
private ToolStripMenuItem alignItem;
```

В конструктор класса `Form1` добавьте новые операторы (листинг 19.4).

Определите обработчик события `Click` для пункта меню `leftJustify1` (листинг 19.5), после чего свяжите созданный обработчик с событием `Click` пунктов меню `center1` и `rightJustify1`.

Листинг 19.3. Настройка свойств

```
пункт меню Left justify (группа Format): Name = leftJustify1,
  ShortcutKeys = Ctrl+L, CheckState = Indeterminate
пункт меню Center (группа Format): Name = center1, ShortcutKeys = Ctrl+E
пункт меню Right justify (группа Format): Name = rightJustify1,
  ShortcutKeys = Ctrl+R
```

Листинг 19.4. Добавление к конструктору формы `Form1`

```
alignItem = leftJustify1;
leftJustify1.Tag = HorizontalAlignment.Left;
center1.Tag = HorizontalAlignment.Center;
rightJustify1.Tag = HorizontalAlignment.Right;
```

Листинг 19.5. Обработчик `leftJustify1.Click`

```
private void leftJustify1_Click(object sender, EventArgs e)
{
  ToolStripMenuItem mi = sender as ToolStripMenuItem;
```

```
    if (mi.Checked) return;
    alignItem.Checked = false;
    alignItem = mi;
    mi.CheckState = CheckState.Indeterminate;
    textBox1.TextAlign = (HorizontalAlignment)mi.Tag;
}
```

Результат. При выполнении добавленных в меню команд устанавливается соответствующее выравнивание текста в редакторе: **Left justify** — по левому краю, **Center** — по центру, **Right justify** — по правому краю. При вызове меню **Format** около команды с текущим выравниванием изображается метка • (как для выбранной радиокнопки). Таким образом, добавленные в данном разделе команды меню ведут себя как *группа переключателей*.

Комментарии

1. Для установки около пункта меню метки • необходимо положить его свойство `CheckState` равным **`Indeterminate`**, однако это действие не приведет к сбросу метки около ранее помеченного пункта меню. Поэтому в класс `Form1` добавлено поле `alignItem`, в котором хранится текущий помеченный пункт меню, связанный с выравниванием. Если выбран другой вариант выравнивания, то с ранее помеченного пункта метка удаляется, на новый пункт метка устанавливается, и этот пункт сохраняется в поле `alignItem` (см. листинг 19.5). Заметим, что свойства `Checked` и `CheckState` связаны между собой: `Checked` равно **`False`** тогда, когда `CheckState` равно **`Unchecked`**; для прочих вариантов `CheckState` свойство `Checked` принимает значение **`True`**.
2. Для того чтобы обеспечить выполнение всех трех команд с помощью одного обработчика, в данном случае мы воспользовались свойством `Tag`, записав в него для каждого пункта меню то значение выравнивания (из перечисления `HorizontalAlignment`), которое соответствует этому пункту. Так как с помощью окна **Properties** свойству `Tag` можно присвоить только значения строкового типа, указанные присваивания выполняются в конструкторе формы (см. листинг 19.4). В обработчике события `Click` (см. листинг 19.5) значение свойства `Tag` выполненной команды присваивается свойству `TextAlign` компонента `textBox1`; при этом выполняется явное преобразование свойства `Tag` к типу перечисления `HorizontalAlignment`. Отметим, что свойствам `Tag` пунктов меню `leftJustify1`, `rightJustify1` и `center1` можно было присвоить числа `0`, `1` и `2`, поскольку такие числовые значения

имеют соответствующие элементы перечисления `HorizontalAlignment`, однако полученный вариант кода был бы менее наглядным:

```
leftJustify1.Tag = 0;
center1.Tag = 2;
rightJustify1.Tag = 1;
```

19.3. Установка цвета символов и фона (команды меню третьего уровня и окно диалога *Цвет*)

Добавьте в форму `Form1` невизуальный компонент типа `ColorDialog` (он получит имя `colorDialog1` и будет размещен в области невизуальных компонентов под изображением формы).

Дополните выпадающее меню, связанное с пунктом **Format** *(см. разд. 19.1 и 19.2)*, добавив в него еще одну горизонтальную разделительную линию и пункт с текстом **`&Colors`**. Затем перейдите в заготовку меню третьего уровня, связанную с пунктом **Colors**, и добавьте в нее пункты меню с текстом **`&Font color...`** и **`&Background color...`**. Настройте свойство `Name` добавленных пунктов меню (листинг 19.6) и определите обработчики события `Click` для пунктов меню `fontColor1` и `backgroundColor1` (листинг 19.7). Вид полученного меню группы **Format** приведен на рис. 19.2 (последний пункт этого меню с именем **Font...** будет добавлен в следующем разделе).

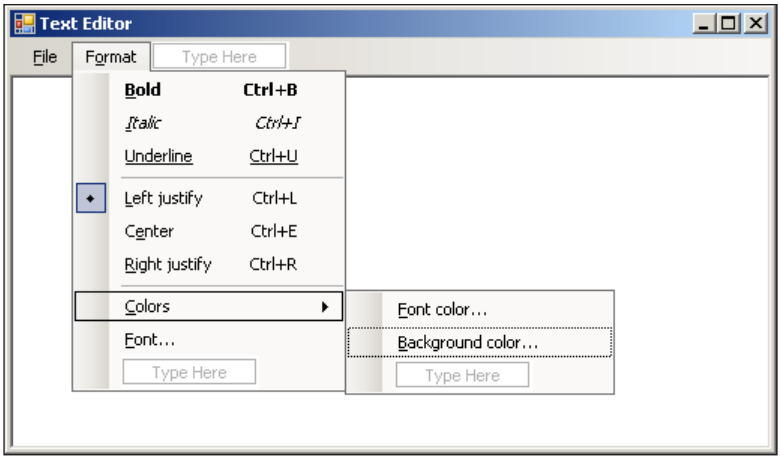


Рис. 19.2. Окончательный вид формы `Form1` для проекта TXTEDIT2

Листинг 19.6. Настройка свойств

```
пункт меню Colors (группа Format): Name = colors1
пункт меню Font color... (группа Colors): Name = fontColor1
пункт меню Background color... (группа Colors): Name = backgroundColor1
```

Листинг 19.7. Обработчики `fontColor1.Click` и `backgroundColor1.Click`

```
private void fontColor1_Click(object sender, EventArgs e)
{
  colorDialog1.Color = textBox1.ForeColor;
  if (colorDialog1.ShowDialog() == DialogResult.OK)
    textBox1.ForeColor = colorDialog1.Color;
}
private void backgroundColor1_Click(object sender, EventArgs e)
{
  colorDialog1.Color = textBox1.BackColor;
  if (colorDialog1.ShowDialog() == DialogResult.OK)
    textBox1.BackColor = colorDialog1.Color;
}
```

Результат. При выполнении команды из группы **Colors** вызывается диалоговое окно **Цвет**, позволяющее установить цвет символов (команда **Font color...**) или цвет фона (команда **Background color...**). При отображении диалогового окна в нем обводится рамкой текущий цвет. Для установки нового цвета надо выбрать его и нажать кнопку **OK**.

> ***ПРИМЕЧАНИЕ***
>
> Если перед выполнением команды меню на экране возникает диалоговое окно, то в конце названия такой команды принято указывать многоточие (ранее в нашей программе многоточие использовалось в названиях команд **Open...** и **Save As...**). Наличие многоточия в названии команды означает, в частности, что данную команду можно отменить уже после ее вызова, если закрыть связанное с ней диалоговое окно с помощью кнопки **Отмена** (Cancel) или клавиши <Esc>.

19.4. Установка свойств шрифта с помощью окна диалога *Шрифт*

Добавьте в форму `Form1` невизуальный компонент типа `FontDialog` (он получит имя `fontDialog1` и будет размещен в области невизуальных компонентов под изображением формы).

Дополните выпадающее меню, связанное с пунктом **Format** *(см. разд. 19.1—19.3)*, добавив в него пункт меню с текстом **&Font...** (см. рис. 19.2). Положите свойство Name добавленного пункта меню равным **font1** и определите для данного пункта меню обработчик события Click (листинг 19.8).

Листинг 19.8. Обработчик font1.Click

```
private void font1_Click(object sender, EventArgs e)
{
  fontDialog1.Font = textBox1.Font;
  if (fontDialog1.ShowDialog() == DialogResult.OK)
  {
    Font f = textBox1.Font;
    textBox1.Font = fontDialog1.Font;
    f.Dispose();
    bold1.Checked = fontDialog1.Font.Bold;
    italic1.Checked = fontDialog1.Font.Italic;
    underline1.Checked = fontDialog1.Font.Underline;
  }
}
```

Результат. При выполнении команды **Font...** вызывается диалоговое окно **Шрифт**, позволяющее изменить шрифт в редакторе. При отображении диалогового окна в нем указываются текущие настройки шрифта. В случае изменения стиля шрифта дополнительно производится корректировка меток для пунктов меню **Bold**, **Italic** и **Underline**. См. также комментарий 1.

Ошибка. Если открыть диалоговое окно **Шрифт** и, *не изменяя никаких свойств шрифта*, закрыть окно, нажав кнопку **OK** или клавишу <Enter>, то при последующем открытии окна **Шрифт** произойдет ошибка времени выполнения. Эта ошибка возникает "внутри" метода ShowDialog и связана со взаимодействием свойств Font объектов fontDialog1 и textBox1, в частности, с удалением старого варианта шрифта для компонента textBox1. К последнему выводу можно прийти, заметив, что если закомментировать оператор f.Dispose(), то указанная ошибка не возникает.

Итак, для исправления ошибки можно было бы не вызывать метод Dispose. Но это плохой вариант исправления, так как он будет приводить к излишнему расходованию ресурсов системы. Разумеется, при отсутствии иных альтернатив таким вариантом можно воспользоваться, однако мы можем выбрать лучший вариант.

Исправление. Добавьте в метод `font1_Click` еще один условный оператор (листинг 19.9).

Листинг 19.9. Новый вариант метода `font1_Click`

```
private void font1_Click(object sender, EventArgs e)
{
  fontDialog1.Font = textBox1.Font;
  if (fontDialog1.ShowDialog() == DialogResult.OK)
    if (!textBox1.Font.Equals(fontDialog1.Font))
    {
      Font f = textBox1.Font;
      textBox1.Font = fontDialog1.Font;
      f.Dispose();
      bold1.Checked = fontDialog1.Font.Bold;
      italic1.Checked = fontDialog1.Font.Italic;
      underline1.Checked = fontDialog1.Font.Underline;
    }
}
```

Результат. Теперь в описанной выше ситуации ошибки времени выполнения не происходит (см. комментарий 2).

ПРИМЕЧАНИЕ

Следует признать, что приведенный вариант настройки шрифта в некоторых (крайне редких) ситуациях все же будет приводить к ошибке времени выполнения. Дело в том, что среди стандартных шрифтов Windows имеются такие, для которых не определены некоторые стили. Например, для шрифта Monotype Corsiva не определен обычный (не курсивный) стиль. Правда, с помощью окна **Шрифт** недопустимые стили для подобных шрифтов установить нельзя, однако это можно попытаться сделать позднее, используя команды настройки стиля шрифта из меню **Format**. При такой попытке и произойдет ошибка времени выполнения. Поскольку подобных шрифтов имеется очень мало, мы не будем предусматривать обработку этой особой ситуации в нашем проекте. Читатель может сам реализовать такую обработку, заключив все операторы метода `bold1_Click` (см. листинг 19.2), *кроме первого*, в `try`-блок, и поместив в разделе `catch` этого `try`-блока вывод сообщения об ошибке "Указанная комбинация стилей недоступна для текущего шрифта", а также оператор восстановления прежнего состояния выбранного пункта меню: `mi.Checked = !mi.Checked`. Отметим также, что с помощью класса `FontFamily` и его метода `IsStyleAvailable` (см. их описание в *разд. 28.1 и 28.3*) можно определить, какие стили доступны для данного шрифта.

Комментарии

1. Для того чтобы определить, установлен ли требуемый стиль у выбранного шрифта, мы воспользовались одноименным логическим свойством класса `Font`: `Bold`, `Italic`, `Underline` (см. листинг 19.8). Напомним, что данные свойства (как и все прочие свойства класса `Font`) доступны только для чтения.
2. Проверка, добавленная в новом варианте метода `font1_Click` (см. листинг 19.9), позволяет обойти все действия по изменению шрифта в том случае, если он не был изменен в диалоговом окне. Метод `Equals` предназначен для проверки того, что два объекта *равны*, т. е. что они имеют одинаковое "значение" (в нашем случае — описывают одинаковые шрифты). Заметим, что операцию `==` в данной ситуации использовать нельзя, так как для многих классов (в том числе для класса `Font`) операция `==` возвращает `true`, только если оба сравниваемых объекта являются *тождественными*, т. е. фактически являются ссылками на *один и тот же объект*, размещенный в памяти. Если попытаться использовать в нашем случае вместо метода `Equals` операцию `==`, можно убедиться в том, что ее результат всегда будет равен `false`, иными словами, после выполнения метода `ShowDialog` свойства `Font` объектов `fontDialog1` и `textBox1` *всегда* будут ссылаться на различные объекты. Таким образом, эти свойства никогда не будут тождественными, однако могут быть равными.

Глава 20

Команды редактирования, контекстное меню: проект TXTEDIT3

Проект TXTEDIT3 продолжает серию проектов, посвященных разработке полнофункционального текстового редактора. В этом проекте реализуются стандартные команды редактирования текста, которые включаются как в главное меню (компонент `MenuStrip`), так и в контекстное меню редактора (компонент `ContextMenuStrip`). Также рассматриваются вопросы, связанные с использованием буфера обмена Windows в .NET-приложениях (класс `Clipboard`).

20.1. Команды редактирования

В качестве заготовки для проекта TXTEDIT3 следует использовать ранее разработанный проект TXTEDIT2 *(см. главу 19)*.

Перейдите в конструктор меню *(см. разд. 18.1)* и вставьте новый пункт меню первого уровня *перед* уже имеющимся пунктом **Format**. Для этого выделите пункт **Format**, вызовите его контекстное меню (нажав правую кнопку мыши) и выберите в нем команду **Insert | MenuItem**. Новый пункт меню сразу получит имя `toolStripMenuItem`*N* (где *N* — некоторое число) и заголовок, совпадающий с этим именем. Заголовок нового пункта меню замените на **`&Edit`** (это можно сделать в самом конструкторе меню), а его имя, т. е. свойство `Name`, замените на **`edit1`** с помощью окна **Properties**.

В выпадающем меню, связанном с новым пунктом **Edit**, создайте пункты меню с текстом **`&Undo`**, дефисом **`-`** (этот пункт будет преобразован в горизонтальную линию), **`Cu&t`**, **`&Copy`**, **`&Paste`**, **`&Delete`**, еще одним дефисом **`-`**, **`&Select All`**. Настройте свойства добавленных пунктов меню (листинг 20.1); в результате меню **Edit** примет вид, приведенный на рис. 20.1.

Определите для пунктов меню **Edit** обработчики события `Click` (листинг 20.2).

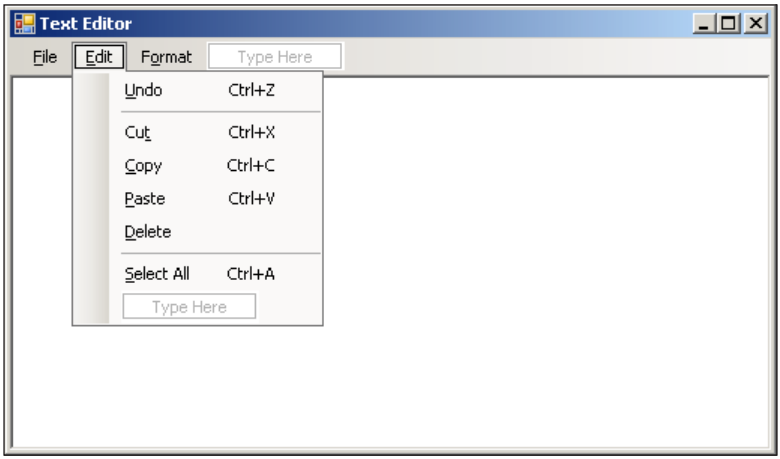


Рис. 20.1. Вид формы `Form1` для проекта TXTEDIT3 с развернутым меню **Edit**

Листинг 20.1. Настройка свойств

пункт меню Undo (группа Edit): Name = **undo1**, ShortcutKeys = **Ctrl+Z**

пункт меню Cut (группа Edit): Name = **cut1**, ShortcutKeys = **Ctrl+X**

пункт меню Copy (группа Edit): Name = **copy1**, ShortcutKeys = **Ctrl+C**

пункт меню Paste (группа Edit): Name = **paste1**, ShortcutKeys = **Ctrl+V**

пункт меню Delete (группа Edit): Name = **delete1**

пункт меню Select All (группа Edit): Name = **selectAll1**,
ShortcutKeys = **Ctrl+A**

Листинг 20.2. Обработчики `undo1.Click`, `cut1.Click`, `copy1.Click`, `paste1.Click`, `delete1.Click`, `selectAll1.Click`

```
private void undo1_Click(object sender, EventArgs e)
{
  textBox1.Undo();
}
private void cut1_Click(object sender, EventArgs e)
{
  textBox1.Cut();
}
private void copy1_Click(object sender, EventArgs e)
{
```

```
  textBox1.Copy();
}
private void paste1_Click(object sender, EventArgs e)
{
  textBox1.Paste();
}
private void delete1_Click(object sender, EventArgs e)
{
  textBox1.SelectedText = "";
}
private void selectAll1_Click(object sender, EventArgs e)
{
  textBox1.SelectAll();
}
```

Результат. В меню определены стандартные команды редактирования: отмена последнего действия **Undo**, вырезание **Cut** и копирование **Copy** выделенного фрагмента в буфер, вставка **Paste** фрагмента из буфера, удаление выделенного фрагмента **Delete**, выделение всего текста **Select All**.

Комментарии

1. Клавиша <Del> является клавишей-ускорителем для команды **Delete** по умолчанию (т. е. она обрабатывается в любом компоненте `TextBox`). В том случае, когда текст не содержит выделенного фрагмента, для клавиши <Del> также предусмотрено определенное действие (удаление символа справа от курсора), поэтому указывать ее в свойстве `delete1.ShortcutKeys` не следует. Комбинации клавиш <Ctrl>+<Z>, <Ctrl>+<X>, <Ctrl>+<C>, <Ctrl>+<V> также обрабатываются нужным образом в любом компоненте `TextBox`. Указав их в соответствующих свойствах `ShortcutKeys`, мы лишь обеспечили отображение этих ускорителей в меню. Все перечисленные команды доступны также из стандартного контекстного меню компонента `TextBox` (контекстное меню вызывается нажатием правой кнопки мыши на компоненте).
2. Следует отметить, что те из реализованных команд редактирования, которые изменяют содержимое компонента `textBox1`, всегда устанавливают его свойство `Modified` равным `true` (в отличие от действий по *программному* изменению свойства `Text`, при выполнении которых свойство `Modified` автоматически получает значение `false`).

20.2. Выделение недоступных команд редактирования. Работа с буфером обмена

Определите обработчик события `Click` для пункта меню `edit1` (листинг 20.3).

Листинг 20.3. Обработчик `edit1.Click`

```
private void edit1_Click(object sender, EventArgs e)
{
  undo1.Enabled = textBox1.CanUndo;
  cut1.Enabled = copy1.Enabled = delete1.Enabled =
    textBox1.SelectionLength > 0;
  paste1.Enabled =
    Clipboard.GetDataObject().GetDataPresent(typeof(string));
  selectAll1.Enabled = textBox1.Text != "";
}
```

Результат. При вызове подменю группы **Edit** недоступные пункты меню выделяются серым цветом. Пункт **Undo** доступен, если ранее было выполнено действие, которое можно отменить. Пункты **Cut**, **Copy** и **Delete** доступны, если текст содержит выделенный фрагмент. Пункт **Paste** доступен, если буфер обмена содержит данные в текстовом формате (см. также комментарий). Пункт **Select All** доступен, если в редакторе содержится непустой текст.

Недочет. Обработчик `edit1_Click` выполняется уже *после* разворачивания подменю команды **Edit**, поэтому выделение недоступных пунктов меню происходит на глазах пользователя, что выглядит неряшливо.

Исправление. Находясь в окне **Properties** в режиме **Events**, выполните для компонента `edit1` следующие действия:

1. Из строки **Click** удалите текст **`edit1_Click`**.
2. В строку **DropDownOpening** добавьте текст **`edit1_Click`** (для этого проще всего воспользоваться выпадающим списком доступных обработчиков).

Результат. Теперь указанные в обработчике действия выполняются после щелчка на пункте **Edit**, но *перед* разворачиванием подменю, поскольку именно в этот момент происходит событие `DropDownOpening`, с которым мы связали обработчик.

Комментарий

Для проверки наличия в буфере обмена текстового фрагмента мы воспользовались статическим методом `GetDataObject` класса `Clipboard`. Этот метод возвращает объект типа `IDataObject`, который позволяет проверить наличие в буфере обмена Windows данных интересующего нас типа с помощью логической функции `GetDataPresent`, а также получить эти данные с помощью функции `GetData`, возвращающей значение типа `object`. Обе функции имеют один параметр типа `Type`, определяющий требуемый тип данных (в нашем случае — `string`). Для проверки того, что в буфере обмена содержится растровый или векторный рисунок, следует вызвать метод `GetDataPresent` с параметром `typeof(Bitmap)` или `typeof(Metafile)` соответственно. Заметим, что буфер обмена может хранить *любые* объекты .NET (например, кнопки), однако получить подобный объект может только .NET-приложение, которое поместило его в буфер (в то же время строки и рисунки, находящиеся в буфере обмена, доступны всем приложениям).

Следует учитывать, что буфер обмена может одновременно хранить данные *разных форматов*, поэтому из того факта, что метод `GetDataPresent(typeof(string))` вернул `true`, не следует, что метод `GetDataPresent(typeof(Bitmap))` обязательно вернет `false`.

Для помещения элемента данных в буфер обмена предназначен статический метод `SetDataObject` класса `Clipboard`. В качестве первого параметра этого метода надо указать объект, содержащий требуемый элемент данных (например, строку). В буфер всегда помещается *копия* элемента данных; при этом из буфера удаляются "старые" данные *того же типа*. Метод `SetDataObject` может иметь второй параметр типа `bool`. Если этот параметр равен `true`, то элемент данных сохранится в буфере обмена и после завершения работы того приложения, которое поместило его в буфер. Понятно, что так следует поступать только для данных, которые могут распознаваться всеми приложениями (например, строк или рисунков). Если второй параметр равен `false` или отсутствует, то при завершении работы приложения элемент данных автоматически удаляется из буфера обмена.

Имеется также возможность размещать в буфере обмена *несколько представлений* одного элемента данных (например, программа, помещающая в буфер текст в формате RTF, может одновременно поместить в него текст, не содержащий форматирования). Приложение может определить, данные в каком формате содержатся в буфере, и загрузить именно тот формат, который требуется. Для этих целей функции `GetDataPresent` и `GetData` снабжены перегруженным вариантом, принимающим в качестве параметра не тип, а текстовую строку с описанием требуемого формата (например, `"Rich Text Format"`). Все воз-

можные форматы представлены в классе DataFormat в виде статических строковых полей, доступных только для чтения.

Подробное описание приемов работы с буфером обмена приводится, например, в главе 24 книги [5].

20.3. Создание контекстного меню

Добавьте в форму Form1 компонент типа ContextMenuStrip (он получит имя contextMenuStrip1 и будет размещен под изображением формы, в области невизуальных компонентов). Данный компонент позволяет использовать в приложении *контекстные меню*. Свяжите компонент contextMenuStrip1 с областью редактирования (компонентом textBox1), положив свойство ContextMenuStrip компонента textBox1 равным **contextMenuStrip1**.

Для определения команд контекстного меню, как и в случае обычного меню, предназначен *конструктор меню*. Однако, в отличие от обычного меню, конструктор для контекстного меню отображается в форме только при выделении компонента, связанного с контекстным меню (в нашем случае — компонента contextMenuStrip1). Горизонтальное меню первого уровня для контекстного меню создавать нельзя; не рекомендуется также создавать в контекстном меню вложенные выпадающие меню и связывать с командами выпадающего меню клавиши-ускорители.

Действуя так же, как в *разд. 18.1* при создании пункта меню **File** и связанных с ним пунктов меню второго уровня, создайте в контекстном меню contextMenuStrip1 пункты меню с текстом **Cu&t**, **&Copy**, **&Paste**, дефис **-** (который будет преобразован в горизонтальную линию), **&Font...** (рис. 20.2). Настройте свойства добавленных пунктов меню (листинг 20.4; обратите внимание на то, что при определении имен для пунктов контекстного меню мы используем не цифру 1, как для пунктов главного меню, а цифру 2). В листинге 20.4 также указано, с какими существующими обработчиками надо связать событие Click для каждой из команд контекстного меню.

Кроме того, определите обработчик события Opening для компонента contextMenuStrip1 (листинг 20.5).

Листинг 20.4. Настройка свойств

```
пункт Cut (меню contextMenuStrip1): Name = cut2, Click = cut1_Click
пункт Copy (меню contextMenuStrip1): Name = copy2, Click = copy1_Click
пункт Paste (меню contextMenuStrip1): Name = paste2, Click = paste1_Click
пункт Font (меню contextMenuStrip1): Name = font2, Click = font1_Click
```

Листинг 20.5. Обработчик `contextMenuStrip1.Opening`

```
private void contextMenuStrip1_Opening(object sender, CancelEventArgs e)
{
  cut2.Enabled = copy2.Enabled = textBox1.SelectionLength > 0;
  paste2.Enabled =
    Clipboard.GetDataObject().GetDataPresent(typeof(string));
}
```

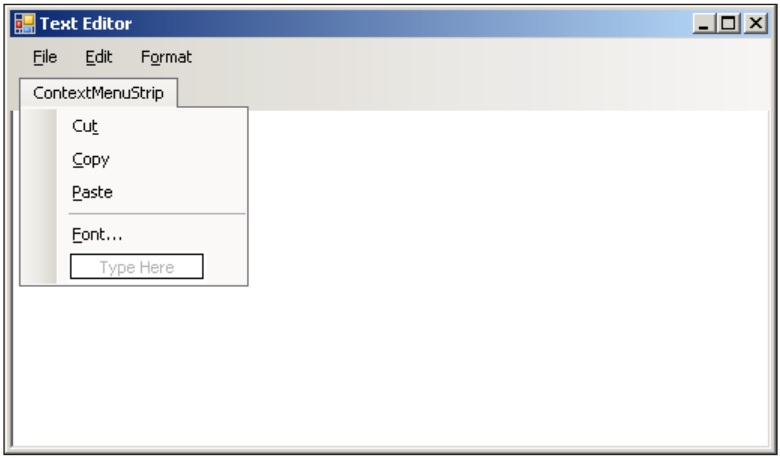


Рис. 20.2. Вид формы `Form1` для проекта TXTEDIT3 с конструктором контекстного меню

Результат. При нажатии правой кнопки мыши в окне редактора вместо стандартного контекстного меню компонента `TextBox` отображается контекстное меню, определенное в компоненте `contextMenuStrip1`. Недоступные в данный момент команды контекстного меню выделяются серым цветом.

Глава 21

Панель инструментов: проект TXTEDIT4

Проект TXTEDIT4 продолжает серию проектов, посвященных разработке полнофункционального текстового редактора. В этом проекте создается и настраивается панель инструментов приложения (компонент `ToolStrip`) и ее элементы: обычные кнопки быстрого доступа, кнопки-флажки и кнопки-переключатели. Описываются действия, необходимые для добавления изображений в файл ресурсов приложения и связывания добавленных изображений с кнопками и командами меню.

21.1. Создание панели инструментов с кнопками быстрого доступа. Добавление изображений к пунктам меню

В качестве заготовки для проекта TXTEDIT4 следует использовать ранее разработанный проект TXTEDIT3 *(см. главу 20)*.

Разместите в форме `Form1` *панель инструментов* — компонент типа `ToolStrip` (этот компонент получит имя `toolStrip1`). Панель инструментов автоматически пристыкуется к верхней границе формы и будет располагаться ниже главного меню `menuStrip1`. Однако она будет заслонять верхнюю часть компонента `textBox1`. Для размещения всех трех компонентов без перекрытий необходимо установить для них правильный *z*-порядок (см. комментарий в *разд. 13.3*).

В нашем случае проще всего выполнить команду **Bring to Front**  для компонента `textBox1`. Заметим, что для этой команды (как и для команды **Send To Back**) предусмотрена не только соответствующая кнопка на панели **Layout**, но и пункт контекстного меню, которое можно вызвать для любого визуального компонента, размещенного в форме.

Помещенная в форму панель инструментов содержит *выпадающий список всех типов компонентов*, которые можно разместить на этой панели. В нашей программе будут использоваться только обычные кнопки (**Button**)

и разделители (**Separator**). При выборе варианта **Button** компонент-кнопка типа `ToolStripButton` размещается на панели и ему сразу присваивается имя (в данном случае `toolStripButton1`). Поскольку кнопки панели инструментов *(кнопки быстрого доступа)* будут связаны с соответствующими командами главного меню, мы будем заменять имя кнопки, предлагаемое по умолчанию, на имя соответствующего пункта меню, дополняя его цифрой `0`, например, `new10`.

Помещенные на панель компоненты автоматически выстраиваются в ряд без промежутков между ними. Для добавления стандартного промежутка-разделителя следует выбрать из списка возможных элементов панели вариант **Separator**. При этом на панель добавляется компонент типа `ToolStripSeparator`, и ему сразу присваивается соответствующее имя (например, `toolStripSeparator1`). Так как в дальнейшем к разделителям обращаться не требуется, мы не будем изменять их имена. Заметим, что взаимное расположение кнопок и разделителей на панели можно изменить, перетаскивая мышью требуемый элемент на новое место.

Добавьте на панель инструментов `toolStrip1` следующие компоненты (в порядке слева направо; имена компонентов-кнопок, т. е. их свойства `Name`, следует указывать в окне **Properties**): кнопка **`new10`**, кнопка **`open10`**, кнопка **`save10`**, разделитель, кнопка **`cut10`**, кнопка **`copy10`**, кнопка **`paste10`**.

Свойство `Text` кнопки быстрого доступа по умолчанию совпадает с ее именем (т. е. свойством `Name`), однако на кнопке оно не отображается, так как предполагается, что кнопка будет содержать не текст, а изображение (за внешний вид кнопки отвечает ее свойство `DisplayStyle`, имеющее по умолчанию значение **`Image`**). Со всеми кнопками, добавленными на панель инструментов, связывается стандартное изображение  . Разумеется, его надо заменить на изображение, соответствующее действию, которое должна выполнять каждая кнопка.

Перед назначением кнопкам изображений следует добавить все необходимые изображения в файл ресурсов разрабатываемого проекта. Опишем действия по добавлению изображений, предполагая, что нам доступна коллекция ресурсов Visual Studio 2005 или 2008.

Выделите первую из добавленных на панель кнопок (`new10`) и в окне **Properties** перейдите к свойству `Image`. Нажмите кнопку с многоточием  рядом с этим свойством, в появившемся окне **Select Resource** выберите переключатель **Project resource file** и нажмите кнопку **Import...**. В появившемся окне **Open** перейдите в подкаталог bitmaps\commands\24bitcolor коллекции VS2005ImageLibrary (или подкаталог Actions\24bitcolor bitmaps коллек-

ции VS2008ImageLibrary), выберите файл Document.bmp и нажмите кнопку **Открыть** или клавишу <Enter>. В результате в окне **Select Resource** появится имя добавленного ресурса: **Document**. Не закрывая окна **Select Resource**, добавьте в файл ресурсов следующие файлы из того же подкаталога: OpenFolder.bmp, Save.bmp, Cut.bmp, Copy.bmp, Paste.bmp.

После добавления всех изображений выберите из их списка вариант **Document** и нажмите кнопку **OK**. Выбранное изображение будет связано с кнопкой быстрого доступа `new10`. Аналогичными действиями следует добавить изображения ко всем кнопкам быстрого доступа. В листинге 21.1 для каждой кнопки указывается имя того изображения из списка ресурсов, которое надо указать в свойстве `Image`. Кроме того, в листинге 21.1 указано, с каким существующим обработчиком надо связать свойство `Click` каждой кнопки. Еще одно свойство, которое следует настроить для каждой кнопки — это `ToolTipText`; данное свойство отвечает за отображение всплывающей подсказки при наведении курсора мыши на кнопку.

Проверьте, что для всех кнопок быстрого доступа свойство `ImageTransparentColor` имеет значение **`Magenta`** (этим цветом во всех использованных изображениях окрашены фрагменты, которые должны быть прозрачными). Вид полученной панели инструментов приводится на рис. 21.1.

После включения изображений в файл ресурсов проекта их можно использовать не только для кнопок быстрого доступа, но и для пунктов главного меню. Для этого достаточно связать соответствующее изображение со свойством `Image` пункта меню (действуя так же, как при связывании изображения с кнопкой). Приведем для каждого пункта меню имя изображения, которое надо с ним связать: `new1` — **`Document`**, `open1` — **`OpenFolder`**, `save1` — **`Save`**, `cut1` — **`Cut`**, `copy1` — **`Copy`**, `paste1` — **`Paste`**. Для всех этих пунктов меню необходимо установить значение свойства `ImageTransparentColor` равным **`Magenta`**.

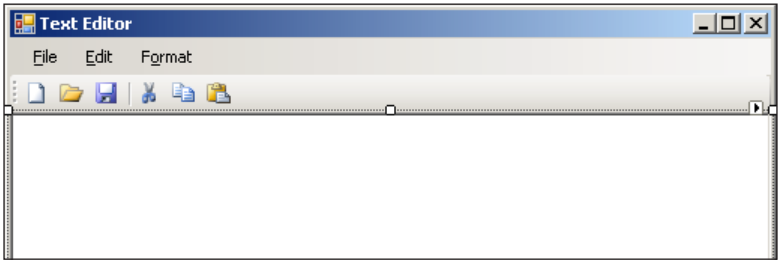


Рис. 21.1. Верхняя часть формы `Form1` для проекта TXTEDIT4 на начальном этапе разработки

Листинг 21.1. Настройка свойств

```
new10: Image = Document, ToolTipText = New, Click = new1_Click
open10: Image = OpenFolder, ToolTipText = Open, Click = open1_Click
save10: Image = Save, ToolTipText = Save, Click = save1_Click
cut10: Image = Cut, ToolTipText = Cut, Click = cut1_Click
copy10: Image = Copy, ToolTipText = Copy, Click = copy1_Click
paste10: Image = Paste, ToolTipText = Paste, Click = paste1_Click
```

Результат. Для выполнения часто используемых команд теперь достаточно щелкнуть мышью на соответствующей кнопке быстрого доступа. Чтобы определить, с какой командой связана кнопка быстрого доступа, надо переместить на нее курсор мыши: через 1—2 секунды около кнопки появится желтый ярлычок с названием соответствующей команды. Около команд меню, имеющих кнопки быстрого доступа, изображаются те же картинки, что и на связанных с ними кнопках.

> ***ПРИМЕЧАНИЕ***
>
> Наличие изображений рядом с командами меню позволяет установить дополнительную визуальную связь между командой меню и соответствующей кнопкой быстрого доступа. Если же с командой меню не связана кнопка, то помещать рядом с ней изображение нецелесообразно.

Комментарий

Для связывания кнопок быстрого доступа с изображениями можно также использовать компонент `ImageList` *(см. разд. 11.7)*. Для этого достаточно разместить в форме данный компонент (который получит имя `imageList1`), добавить в него требуемые изображения и положить свойство `ImageList` панели инструментов `toolStrip1` равным **`imageList1`**. Для кнопок быстрого доступа останется задать свойства `ImageKey` (подобно тому, как это было сделано для "обычной" кнопки `Button` в *разд. 11.7*). К сожалению, в интегрированных средах Visual C# 2005 и 2008 в окне **Properties** *отсутствуют* все свойства компонентов `ToolStrip` и `ToolStripButton`, связанные с доступом к компоненту `ImageList`. Поэтому при использовании подобного варианта связывания кнопок с изображениями программист вынужден задавать требуемые свойства *явным образом* в конструкторе формы; кроме того, придется смириться с тем, что в режиме дизайна требуемые изображения на кнопках будут отсутствовать.

21.2. Размещение на панели инструментов кнопок-флажков и кнопок-переключателей

Для того чтобы рассмотреть особенности, связанные с выводом на кнопках быстрого доступа не изображений, а текста, все кнопки, добавленные на панель инструментов в данном разделе, будут снабжены *текстовыми подписями*. Эти подписи указываются в свойстве `Text`. Для отображения на кнопке текстовой подписи свойство `DisplayStyle` кнопки необходимо положить равным **`Text`**.

Добавьте на панель инструментов `toolStrip1` новые компоненты (в порядке слева направо; имена компонентов-кнопок, т. е. их свойства `Name`, следует указывать в окне **Properties**): разделитель, кнопка **`bold10`**, кнопка **`italic10`**, кнопка **`underline10`**, разделитель, кнопка **`leftJustify10`**, кнопка **`center10`**, кнопка **`rightJustify10`**. Настройте свойства добавленных компонентов (листинг 21.2). Вид полученной панели инструментов приводится на рис. 21.2.

В класс `Form1` добавьте вспомогательный метод `GetButton` (листинг 21.3) и определите обработчики события `Click` для новых кнопок быстрого доступа (листинг 21.4).

Измените методы `bold1_Click`, `leftJustify1_Click` и `font1_Click` (листинг 21.5).

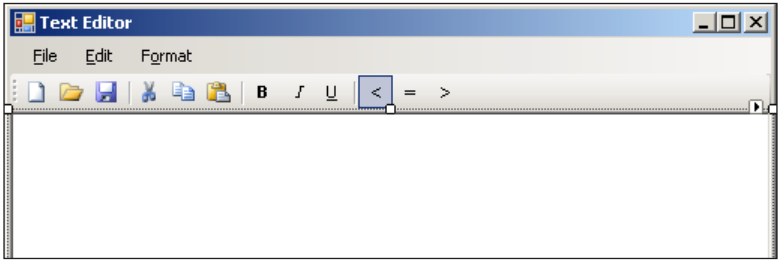


Рис. 21.2. Окончательный вариант верхней части формы `Form1` для проекта TXTEDIT4

Листинг 21.2. Настройка свойств

```
bold10: Text = B, DisplayStyle = Text, ToolTipText = Bold,
  Font.Bold = True
italic10: Text = I, DisplayStyle = Text, ToolTipText = Italic,
  Font.Italic = True
underline10: Text = U, DisplayStyle = Text, ToolTipText = Underline,
  Font.Underline = True
leftJustify10: Text = <, DisplayStyle = Text, ToolTipText = Left justify,
```

```
  Checked = True
center10: Text = =, DisplayStyle = Text, ToolTipText = Center
rightJustify10: Text = >, DisplayStyle = Text,
  ToolTipText = Right justify
```

Листинг 21.3. Метод GetButton формы Form1

```
private ToolStripButton GetButton(ToolStripMenuItem mi)
{
  return toolStrip1.Items[mi.Name + "0"] as ToolStripButton;
}
```

Листинг 21.4. Обработчики bold10.Click, italic10.Click, underline10.Click, leftJustify10.Click, center10.Click, rightJustify10.Click

```
private void bold10_Click(object sender, EventArgs e)
{
  bold1_Click(bold1, null);
}
private void italic10_Click(object sender, EventArgs e)
{
  bold1_Click(italic1, null);
}
private void underline10_Click(object sender, EventArgs e)
{
  bold1_Click(underline1, null);
}
private void leftJustify10_Click(object sender, EventArgs e)
{
  leftJustify1_Click(leftJustify1, null);
}
private void center10_Click(object sender, EventArgs e)
{
  leftJustify1_Click(center1, null);
}
private void rightJustify10_Click(object sender, EventArgs e)
{
  leftJustify1_Click(rightJustify1, null);
}
```

Листинг 21.5. Новый вариант методов bold1_Click, leftJustify1_Click и font1_Click

```
private void bold1_Click(object sender, EventArgs e)
{
  ToolStripMenuItem mi = sender as ToolStripMenuItem;
  mi.Checked = !mi.Checked;
  FontStyle fs = textBox1.Font.Style;
  fs = mi.Checked ? (fs | mi.Font.Style) : (fs & ~mi.Font.Style);
  Font f = textBox1.Font;
  textBox1.Font = new Font(f, fs);
  f.Dispose();
  ToolStripButton sb = GetButton(mi);
  sb.Checked = !sb.Checked;
}
private void leftJustify1_Click(object sender, EventArgs e)
{
  ToolStripMenuItem mi = sender as ToolStripMenuItem;
  if (mi.Checked) return;
  GetButton(alignItem).Checked = alignItem.Checked = false;
  alignItem = mi;
  mi.CheckState = CheckState.Indeterminate;
  GetButton(mi).Checked = true;
  textBox1.TextAlign = (HorizontalAlignment)mi.Tag;
}
private void font1_Click(object sender, EventArgs e)
{
  fontDialog1.Font = textBox1.Font;
  if (fontDialog1.ShowDialog() == DialogResult.OK)
    if (!textBox1.Font.Equals(fontDialog1.Font))
    {
      Font f = textBox1.Font;
      textBox1.Font = fontDialog1.Font;
      f.Dispose();
      bold10.Checked = bold1.Checked = fontDialog1.Font.Bold;
      italic10.Checked = italic1.Checked = fontDialog1.Font.Italic;
      underline10.Checked = underline1.Checked =
        fontDialog1.Font.Underline;
    }
}
```

Результат. Для настройки стиля шрифта и выравнивания текста достаточно нажать соответствующую кнопку быстрого доступа, переведя ее тем самым в "нажатое" состояние. Далее мы будем говорить о "нажатом" и "отжатом" состоянии кнопки, хотя при использовании стандартного стиля изображения кнопок в .NET "нажатая" кнопка просто выделяется рамкой и фоновым цветом (см. изображение кнопки < на рис. 21.2).

Нажатое состояние кнопки быстрого доступа означает, что указанный режим включен. Кнопки настройки шрифта `bold10`, `italic10`, `underline10` действуют как *флажки*: каждую кнопку можно перевести в нажатое или отжатое состояние независимо от остальных. Кнопки выравнивания текста `leftJustify10`, `center10`, `rightJustify10` действуют как *группа переключателей*: нажатие на любую из них приводит к освобождению (отжатию) остальных. Следует отметить, что при выполнении команд форматирования непосредственно из меню или с помощью клавиш-ускорителей состояние соответствующих кнопок быстрого доступа корректируется требуемым образом.

Комментарии

1. Добавленный нами к классу `Form1` метод `GetButton` (см. листинг 21.3) позволяет по пункту меню определить связанную с ним кнопку быстрого доступа. Это возможно, благодаря выбору похожих имен для пунктов меню и связанных с ними кнопок: для получения кнопки достаточно использовать свойство-коллекцию `Items` компонента `toolStrip1`, указав в качестве ключа имя нужной кнопки, а это имя равно имени соответствующей команды меню, дополненному символом `0`. Заметим, что метод `GetButton` позволяет также проверить, связана ли с данным пунктом меню какая-либо кнопка быстрого доступа: если с пунктом не связана кнопка, то метод вернет `null` (в нашей программе эта дополнительная возможность не используется, но она может оказаться полезной при реализации взаимодействия пунктов меню и кнопок быстрого доступа в других приложениях).
2. В приведенной программной реализации кнопки "делегируют" выполнение всех действий (в том числе изменение их собственного состояния) командам меню. Это гарантирует, что состояние кнопок будет правильно изменено и при выполнении команд форматирования другими способами (из меню или с помощью клавиш-ускорителей). Для реализации поведения компонентов `leftJustify10`, `center10`, `rightJustify10` как группы переключателей используется поле `alignItem`, с помощью которого снимается выделение с ранее выделенной кнопки данной группы (см. листинг 21.5).

Глава 22

Статусная панель и подсказки: проект TXTEDIT5

Проект TXTEDIT5 продолжает серию проектов, посвященных разработке полнофункционального текстового редактора. В этом проекте создается и настраивается статусная панель приложения (компонент `StatusStrip`) и ее элементы-метки. Описываются действия, обеспечивающие проверку состояния клавиш-переключателей <Caps Lock> и <Num Lock>, скрытие и повторное отображение панели инструментов и статусной панели, а также вывод развернутой подсказки к выделенной команде меню на статусную панель. Кроме того, описывается компонент `ToolTip`, позволяющий связывать контекстную подсказку с любым визуальным компонентом приложения.

22.1. Использование статусной панели

В качестве заготовки для проекта TXTEDIT5 следует использовать ранее разработанный проект TXTEDIT4 *(см. главу 21)*.

Добавьте в форму `Form1` *статусную панель* — компонент типа `StatusStrip` (этот компонент получит имя `statusStrip1`), а также невизуальный компонент типа `Timer` (который получит имя `timer1` и будет указан в области невизуальных компонентов под изображением формы). Для того чтобы нижняя часть компонента `textBox1` не была перекрыта добавленной статусной панелью, следует выполнить команду **Bring to Front**  для компонента `textBox1` *(см. разд. 21.1)*.

Для добавления компонентов на статусную панель предназначен выпадающий список, в котором мы будем использовать только вариант **StatusLabel** (этот вариант добавляет на статусную панель компонент-метку типа `ToolStripStatusLabel`). Поскольку имена по умолчанию для добавленных на панель компонентов являются чрезмерно длинными (например, `toolStripStatusLabel1`), мы будем изменять их, выбирая имена в соответствии с назначением каждого элемента статусной строки.

Добавьте на статусную панель `statusStrip1` компоненты-метки с указанными именами (в порядке слева направо; имена компонентов-меток, т. е. их свойства `Name`, следует указывать в окне **Properties**): **`cap1`**, **`num1`**, **`modified1`**, **`hint1`**. Настройте свойства добавленных компонентов-меток, а также компонента `timer1` (листинг 22.1; свойство `Size.Width` используется для задания новой ширины метки, а свойство `Margin.Left` метки `hint1` — для увеличения промежутка между текстом **Ready** этой метки и рамкой предыдущей метки `modified1`). Вид полученной статусной панели приводится на рис. 22.1.

Определите обработчик события `Tick` для компонента `timer1` (листинг 22.2) и добавьте вызов этого обработчика в конструктор класса `Form1`:

```
timer1_Tick(this, null);
```

Рис. 22.1. Вид нижней части формы `Form1` для проекта TXTEDIT5

Листинг 22.1. Настройка свойств

```
cap1: Text = CAP, AutoSize = False, BorderSides = All,
  BorderStyle = Sunken, Size.Width = 40
num1: Text = NUM, AutoSize = False, BorderSides = All,
  BorderStyle = Sunken, Size.Width = 40
modified1: Text = Modified, AutoSize = False, BorderSides = All,
  BorderStyle = Sunken, Size.Width = 60
hint1: Text = Ready, Margin.Left = 5
timer1: Interval = 200, Enabled = True
```

Листинг 22.2. Обработчик `timer1.Tick`

```
private void timer1_Tick(object sender, EventArgs e)
{
  cap1.Text = Control.IsKeyLocked(Keys.CapsLock) ? "CAP" : "";
  num1.Text = Control.IsKeyLocked(Keys.NumLock) ? "NUM" : "";
  modified1.Text = textBox1.Modified ? "Modified" : "";
}
```

Результат. На статусной панели отображается текущее состояние клавиш <Caps Lock> и <Num Lock>. Кроме того, в ней показывается, изменялся ли

текст в области редактирования после его последнего сохранения (если изменялся, то на статусной панели выводится строка "Modified").

Комментарии

1. Для возможности определения текущего состояния клавиш <Caps Lock>, <Num Lock> и <Scroll Lock> в класс `Control` в версии .NET 2.0 был добавлен новый статический метод `IsKeyLocked` логического типа. В качестве параметра этого метода можно указывать только три элемента перечисления `Keys`: `Keys.CapsLock`, `Keys.NumLock` и `Keys.Scroll`; попытка указать другой элемент перечисления `Keys` приводит к возбуждению исключения `NotSupportedException`.
2. Для отслеживания изменения свойства `Modified` можно также использовать событие `ModifiedChanged` компонента `textBox1`.
3. Вызов обработчика события `Tick` в конструкторе формы обеспечивает настройку вида статусной панели перед ее отображением на экране.

22.2. Недоступные кнопки быстрого доступа

Добавьте в метод `timer1_Tick` новые операторы (листинг 22.3).

Листинг 22.3. Добавление к методу `timer1_Tick`

```
cut10.Enabled = copy10.Enabled = textBox1.SelectionLength > 0;
paste10.Enabled =
  Clipboard.GetDataObject().GetDataPresent(typeof(string));
save1.Enabled = save10.Enabled = textBox1.Modified;
```

Результат. Теперь состояние кнопок быстрого доступа, связанных с буфером обмена, совпадает с состоянием соответствующих команд меню (они одновременно доступны или недоступны). Кроме того, команда **Save** и связанная с ней кнопка быстрого доступа теперь доступны только после внесения изменений в редактируемый текст.

22.3. Скрытие панелей

Действуя так же, как в *разд. 18.1* при создании пункта меню **File** и связанных с ним пунктов меню второго уровня, добавьте в компонент `menuStrip1` пункт меню первого уровня с текстом **`&View`** и с помощью окна **Properties** измените имя этого пункта (т. е. свойство `Name`) на **`view1`**. В выпадающем меню,

связанном с пунктом **View**, создайте пункты с текстом **`&Tool bar`** и **`&Status bar`** (рис. 22.2). Настройте свойства добавленных пунктов меню (листинг 22.4).

Определите обработчики события `Click` для пунктов меню `toolBar1` и `statusBar1` (листинг 22.5).

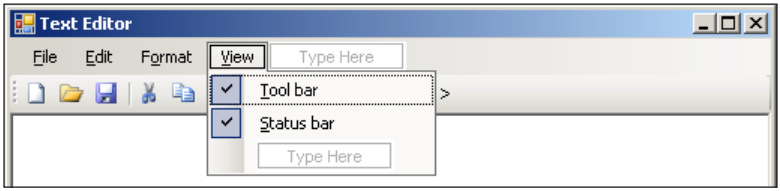


Рис. 22.2. Вид верхней части `Form1` для проекта TXTEDIT5

Листинг 22.4. Настройка свойств

```
пункт меню Tool bar (группа View): Name = toolBar1, Checked = True
пункт меню Status bar (группа View): Name = statusBar1, Checked = True
```

Листинг 22.5. Обработчики `toolbar1.Click` и `statusBar1.Click`

```
private void toolBar1_Click(object sender, EventArgs e)
{
  toolStrip1.Visible = toolBar1.Checked = !toolBar1.Checked;
}
private void statusBar1_Click(object sender, EventArgs e)
{
  statusStrip1.Visible = statusBar1.Checked = !statusBar1.Checked;
}
```

Результат. С помощью команд-переключателей меню **View** можно скрывать и восстанавливать панель инструментов и статусную панель.

22.4. Вывод подсказок на статусную панель

Настройте свойство `ToolTipText` для пунктов меню группы **File** (листинг 22.6).

Определите обработчики событий `MouseEnter` и `MouseLeave` для пункта меню `new1` (листинг 22.7), после чего свяжите эти обработчики с событиями

MouseEnter и MouseLeave остальных пунктов меню группы **File**. Обработчик new1_MouseLeave свяжите также с событием MenuActivate компонента menuStrip1.

Кроме того, определите обработчик события MenuDeactivate для компонента menuStrip1 (листинг 22.8).

Листинг 22.6. Настройка свойств

```
new1: ToolTipText = Создать новый документ
open1: ToolTipText = Открыть существующий документ
save1: ToolTipText = Сохранить текущий документ
saveAs1: ToolTipText = Сохранить документ под новым именем
exit1: ToolTipText = Выйти из редактора
```

Листинг 22.7. Обработчики new1.MouseEnter и new1.MouseLeave

```
private void new1_MouseEnter(object sender, EventArgs e)
{
  hint1.Text = (sender as ToolStripMenuItem).ToolTipText;
}
private void new1_MouseLeave(object sender, EventArgs e)
{
  hint1.Text = "";
}
```

Листинг 22.8. Обработчик menuStrip1.MenuDeactivate

```
private void menuStrip1_MenuDeactivate(object sender, EventArgs e)
{
  hint1.Text = "Ready";
}
```

Результат. При перемещении курсора мыши над пунктами меню **File** на статусной панели выводится подсказка к выделенной команде (если данная команда является доступной). Для прочих команд меню подсказка на статусной панели не отображается. При выходе из меню на статусной панели восстанавливается текст "Ready".

Недочет. При наведении мыши на пункт меню из группы **File** около данного пункта возникает дополнительная всплывающая подсказка, текст которой совпадает с текстом, выведенным на статусную панель.

Исправление. В конструктор класса `Form1` добавьте оператор, подавляющий отображение всплывающих подсказок для выпадающего меню **File**:

```
file1.DropDown.ShowItemToolTips = false;
```

> ***ПРИМЕЧАНИЕ***
>
> Настроить свойство `ShowItemToolTips` для *главного меню* можно в окне свойств для компонента `menuStrip1` (заметим, что по умолчанию это свойство равно **`False`**). Однако настроить его для *выпадающих подменю* (объектов типа `ToolStripDropDown`) можно только программным путем.

Комментарии

1. Как отмечено в главе 20 книги [5], "одна из основных задач строки состояния — отображение подсказок для меню". К сожалению, следует признать, что средства для реализации этой основной задачи в компонентах `MenuStrip` и `ToolStripMenuItem` практически отсутствуют. Приведенный в данном разделе вариант реализации является половинчатым, поскольку не позволяет получать подсказки для пунктов меню, перемещаясь по ним с помощью *клавиатуры*. Интересно отметить, что у предшественника класса `ToolStripMenuItem` — класса `MenuItem`, предназначенного для использования совместно со "старым" меню типа `MainMenu`, — имелось событие `Select`, которое лучше подходило для решения поставленной задачи, поскольку реагировало на выделение пункта меню, выполненное любым способом: как мышью, так и клавиатурой (впрочем, события, связанного с потерей выделения, не было предусмотрено, поэтому для корректной смены подсказок требовалось снабдить обработчиками события `Select` *все* пункты меню). Ничего аналогичного событию `Select` нет ни в классе `ToolStripMenuItem`, ни в классе `MenuStrip`.
2. У большинства "обычных" визуальных компонентов (не связанных с меню, панелями инструментов и статусными панелями) нет свойства, аналогичного `ToolTipText`. Тем не менее, для любого визуального компонента можно реализовать появление всплывающей подсказки, если воспользоваться невизуальным компонентом `ToolTip`. Помимо большого количества свойств, позволяющих настроить вид отображаемой подсказки, компонент `ToolTip` имеет метод `SetToolTip(control, hint)`, связывающий стро-

ковую подсказку `hint` с визуальным компонентом `control` (парный метод `GetToolTip(control)` позволяет получить значение подсказки, связанной с компонентом `control`). После добавления компонента `ToolTip` в форму у всех визуальных компонентов формы в окне **Properties** будет отображаться свойство вида `ToolTip on toolTip1`, позволяющее легко задавать строки подсказок в режиме дизайна. С помощью события `Popup` компонента `ToolTip` можно перенаправлять вывод текста подсказок на другие компоненты (например, на статусную панель).

Глава 23

Форматирование документа: проект TXTEDIT6

Проект TXTEDIT6 завершает серию проектов, посвященных разработке полнофункционального текстового редактора. В этом проекте описываются действия по замене компонента `TextBox` на компонент `RichTextBox`, позволяющий по-разному форматировать различные фрагменты текста. Рассматриваются возможности компонента `RichTextBox`, связанные с форматированием шрифта и абзаца. Кроме того, описываются средства, позволяющие определить позицию курсора в тексте и сохранить текст в различных форматах.

23.1. Замена компонента *TextBox* на компонент *RichTextBox*

В качестве заготовки для проекта TXTEDIT6 следует использовать ранее разработанный проект TXTEDIT5 *(см. главу 22)*.

Для того чтобы заменить уже имеющийся в форме `Form1` компонент `textBox1` (типа `TextBox`) на компонент типа `RichTextBox`, обладающий более богатыми возможностями форматирования, достаточно немного откорректировать файл Form1.Designer.cs. Напомним, что в этом файле содержится информация, добавленная в проект в ходе работы с дизайнером формы (в том числе с дизайнером меню) и окном **Properties**, поэтому обычно не возникает необходимости в его явном редактировании. Однако ничто не запрещает вносить в файл Form1.Designer.cs изменения; следует только учитывать, что именно из этого файла берутся сведения, используемые для отображения формы в режиме дизайна.

Загрузите в редактор файл Form1.Designer.cs (проще всего это сделать с помощью окна **Solution Explorer**), найдите в нем строку

```
private System.Windows.Forms.TextBox textBox1;
```

и измените ее следующим образом:

```
private System.Windows.Forms.RichTextBox textBox1;
```

Тем самым мы изменили тип компонента `textBox1`. Имя компонента изменять не будем, так как оно неоднократно используется в имеющемся коде нашей программы.

Необходимо также изменить оператор, который создает компонент `textBox1`. Данный оператор содержится в том же файле Form1.Designer.cs, причем в разделе **Windows Form Designer generated code**, который по умолчанию является скрытым. Разверните этот раздел, щелкнув мышью на знаке + слева от его заголовка, и найдите в нем оператор

```
this.textBox1 = new System.Windows.Forms.TextBox();
```

Для нахождения указанного оператора можно, например, организовать поиск слова **textBox1** (о режимах поиска, имеющихся в среде Visual C#, см. комментарий 2 в *разд. 10.7*). Измените найденный оператор следующим образом:

```
this.textBox1 = new System.Windows.Forms.RichTextBox();
```

После внесения указанных изменений файл Form1.Designer.cs можно закрыть, щелкнув правой кнопкой мыши на его ярлычке и выбрав в появившемся контекстном меню команду **Close**.

Компонент `RichTextBox` позволяет устанавливать различные форматные настройки для разных фрагментов текста. Эти настройки сохраняются вместе с текстом в специальном формате, называемом *Rich Text Format* (формат RTF). Файлы, содержащие текст в данном формате, обычно имеют расширение rtf. Внесите в программу изменения, позволяющие использовать дополнительные возможности компонента `RichTextBox`.

Для компонентов `saveFileDialog1` и `openFileDialog1` измените значения свойства `DefaultExt` на **`rtf`**, а свойства `Filter` на **`RTF files|*.rtf`**.

Измените методы `SaveToFile` и `open1_Click` класса `Form1` (листинг 23.1).

В методах `bold1_Click` и `font1_Click` (листинг 21.5) замените все фрагменты `textBox1.Font` на **`textBox1.SelectionFont`** (в `bold1_Click` таких фрагментов должно быть три, в `font1_Click` — четыре).

В методе `fontColor1_Click` (листинг 19.7) замените все фрагменты `textBox1.ForeColor` на **`textBox1.SelectionColor`** (таких фрагментов должно быть два).

В методе `backgroundColor1_Click` (листинг 19.7) замените все фрагменты `textBox1.BackColor` на **`textBox1.SelectionBackColor`** (таких фрагментов должно быть два).

В методе `leftJustify1_Click` (листинг 21.5) замените в последнем операторе фрагмент `textBox1.TextAlign` на **`textBox1.SelectionAlignment`**.

Наконец, установите свойство EnableAutoDragDrop компонента textBox1 равным **True**.

Листинг 23.1. Новый вариант методов SaveToFile и open1_Click

```
private void SaveToFile(string path)
{
  textBox1.SaveFile(path, RichTextBoxStreamType.RichText);
  textBox1.Modified = false;
}
private void open1_Click(object sender, EventArgs e)
{
  if (TextSaved())
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
      string path = openFileDialog1.FileName;
      textBox1.LoadFile(path, RichTextBoxStreamType.RichText);
      Text = "Text Editor - " + Path.GetFileName(path);
      saveFileDialog1.FileName = path;
      openFileDialog1.FileName = "";
    }
}
```

Результат. Теперь команды форматирования шрифта влияют на выделенный фрагмент текста, а если выделенного фрагмента нет, то на последующие вводимые символы. Команды выравнивания абзаца влияют на выделенные абзацы (в том числе и на абзацы, в которых выделена только часть текста), а если выделенных абзацев нет, то на текущий абзац. В файле можно сохранять текст вместе с форматными настройками (расширение файла по умолчанию — rtf). При загрузке файлов в формате RTF на экране отображается текст вместе с сохраненными форматными настройками.

Если текст не умещается по высоте в окне редактора, то справа появляется вертикальная полоса прокрутки (подобная возможность *автоматического* отображения вертикальной полосы прокрутки у компонента TextBox отсутствует). Слишком большие по ширине строки, как и ранее, автоматически переносятся на новую строку (см. комментарий 5 в *разд. 18.3*).

Благодаря значению **True** свойства EnableAutoDragDrop, появившегося у компонента RichTextBox в версии .NET 2.0, в редакторе обеспечивается режим

перетаскивания drag & drop для выделенных фрагментов текста: если зацепить мышью выделенный фрагмент, то его можно перетащить на новое место, а если при перетаскивании держать нажатой клавишу <Ctrl> (при этом около курсора мыши будет изображен символ +), то произойдет не перемещение, а *копирование* выделенного фрагмента.

Недочеты. При перемещении курсора на участок текста с иным форматированием состояние кнопок быстрого доступа и команд форматирования в меню **Format** не меняется; при загрузке нового файла атрибут `Modified` не сбрасывается; при выполнении команд **New** и **Open** не происходит корректировки форматных настроек в меню и на панели инструментов.

Помимо этих легко выявляемых недочетов, исправленный вариант нашей программы содержит *ошибку*, которую заметить не так просто. Для того чтобы обнаружить эту ошибку, необходимо выделить фрагмент текста, содержащего *более одного вида шрифта* (например, шрифты Microsoft Sans Serif и Arial) и попытаться выполнить одну из команд настройки стиля шрифта (**Bold**, **Italic** или **Underline**) или установить для фрагмента новый вид шрифта с помощью команды **Font**. В подобной ситуации будет возбуждено исключение `NullReferenceException`.

Отмеченные недочеты и ошибки программы будут исправлены в следующем разделе.

Комментарии

1. При корректировке методов, связанных с сохранением и загрузкой файлов (см. листинг 23.1), были использованы методы `SaveFile` и `LoadFile` компонента `RichTextBox` (у компонента `TextBox` аналогичные методы отсутствуют).
2. При корректировке методов, связанных с форматированием, были использованы свойства `SelectionFont`, `SelectionColor`, `SelectionBackColor` и `SelectionAlignment` компонента `RichTextBox`. Все эти свойства позволяют определять и изменять форматные настройки выделенного фрагмента, а при его отсутствии — позиции текста, в которой находится курсор. Первое свойство отвечает за характеристики шрифта, второе — за цвет символов, третье — за цвет фона, четвертое — за горизонтальное выравнивание абзаца. Некоторые другие свойства, связанные с форматированием абзацев, будут использованы в *разд. 23.3*.

23.2. Корректировка состояния кнопок быстрого доступа и команд меню при изменении текущего формата

В класс `Form1` добавьте новый метод `SetEnabled` (листинг 23.2) и определите обработчик события `SelectionChanged` для компонента `textBox1` (листинг 23.3).

В метод `new1_Click` (см. листинг 18.9) после оператора

```
saveFileDialog1.FileName = "";
```

вставьте оператор

```
textBox1_SelectionChanged(this, null);
```

В метод `open1_Click` (см. листинг 23.1) после оператора

```
openFileDialog1.FileName = "";
```

вставьте операторы

```
textBox1.Modified = false;
textBox1_SelectionChanged(this, null);
```

Листинг 23.2. Метод `SetEnabled` формы `Form1`

```
private void SetEnabled(bool value)
{
  bold1.Enabled = bold10.Enabled =
    italic1.Enabled = italic10.Enabled =
    underline1.Enabled = underline10.Enabled =
    font1.Enabled = value;
}
```

Листинг 23.3. Обработчик `textBox1.SelectionChanged`

```
private void textBox1_SelectionChanged(object sender, EventArgs e)
{
  Font f = textBox1.SelectionFont;
  SetEnabled(f != null);
  if (f != null)
  {
    bold1.Checked = bold10.Checked = f.Bold;
```

```
    italic1.Checked = italic10.Checked = f.Italic;
    underline1.Checked = underline10.Checked = f.Underline;
  }
  ToolStripMenuItem mi = leftJustify1;
  switch (textBox1.SelectionAlignment)
  {
    case HorizontalAlignment.Center:
      mi = center1; break;
    case HorizontalAlignment.Right:
      mi = rightJustify1; break;
  }
  if (mi == alignItem) return;
  alignItem.Checked = GetButton(alignItem).Checked = false;
  mi.CheckState = CheckState.Indeterminate;
  GetButton(mi).Checked = true;
  alignItem = mi;
}
```

Результат. Исправлены все недочеты, отмеченные в *разд. 23.1*. В частности, теперь состояние кнопок быстрого доступа и команд меню соответствует формату текущей позиции текста.

Если в тексте выделен фрагмент, то вид команд меню и кнопок быстрого доступа зависит от того, содержит ли выделенный фрагмент участки с различным форматированием. Если во фрагменте используется несколько видов шрифтов, то команды настройки шрифта (**Bold**, **Italic**, **Underline** и **Font**) и связанные с ними кнопки *становятся недоступными* (это позволяет избежать ошибки, отмеченной в *разд. 23.1*). Если во фрагменте используется один вид шрифта, то указанные команды являются доступными, причем команды настройки стиля шрифта **Bold**, **Italic**, **Underline** и связанные с ними кнопки выделяются только в случае, если *весь* выделенный фрагмент имеет соответствующий стиль шрифта (например, кнопка **Bold** будет выделена только в случае, если для всего фрагмента установлен полужирный стиль).

Аналогичным образом ведут себя команды, связанные с выравниванием текста: если для всего фрагмента установлен одинаковый способ выравнивания, то выделяется команда и кнопка, соответствующие этому способу выравнивания; если же выделенный фрагмент содержит абзацы с *различными* вариантами выравнивания, то выделяется команда **Left justify** и связанная с ней кнопка.

Комментарий

При использовании свойства SelectionFont необходимо учитывать его важную особенность: если в выделенном фрагменте имеется более одного вида шрифта, то свойство SelectionFont равно null. В подобной ситуации попытка обращения к любому члену свойства SelectionFont (например, к его свойствам Bold, Italic или Underline) приведет к возбуждению исключения NullReferenceException. Поэтому в методе textBox1_SelectionChanged обращение к членам свойства SelectionFont производится только в случае, если оно не равно null. Однако в нашей программе имеются еще два метода, в которых производится обращение к членам свойства SelectionFont: это новые варианты методов bold1_Click и font1_Click. Чтобы избежать возможной ошибки при выполнении методов bold1_Click и font1_Click, все команды меню и кнопки быстрого доступа, вызывающие эти методы, делаются недоступными в случае, если в тексте выделяется фрагмент, содержащий более одного вида шрифта (для этого в обработчике textBox1_SelectionChanged вызывается вспомогательный метод SetEnabled, приведенный в листинге 23.2).

Заметим, что для команд выравнивания текста опасности ошибки не возникает: если выделенный фрагмент содержит несколько вариантов выравнивания, то свойство SelectionAlignment возвращает значение HorizontalAlignment.Left. Таким образом, в подобной ситуации будет выделена кнопка и команда меню, соответствующая выравниванию по левой границе, что представляется вполне естественным. Не возникает проблем и для команд настройки цвета символов и фона *(см. разд. 19.3)*: если при выполнении этих команд в тексте имеется выделенный фрагмент с различными вариантами цветовой настройки, то при отображении диалогового окна выбора цвета в нем обводится рамкой *черный* цвет.

23.3. Настройка свойств абзаца

Добавьте к проекту новую форму Form2 и разместите в ней компонент-контейнер типа GroupBox (он получит имя groupBox1). В компоненте groupBox1 разместите три метки (label1, label2 и label3) и три компонента типа NumericUpDown (numericUpDown1, numericUpDown2 и numericUpDown3). Кроме того, разместите в форме Form1 еще одну метку (label4), выпадающий список (comboBox1), флажок (checkBox1) и две кнопки (button1 и button2).

Настройте свойства формы Form2 и ее компонентов (листинг 23.4) и расположите компоненты в соответствии с рис. 23.1. Напомним, что настройка Modifiers = **Internal** позволяет обратиться к компоненту из другой формы этого же проекта.

Дополнительно определите свойство `Items` компонента `comboBox1`. Для этого свойства предусмотрено специальное диалоговое окно (см. рис. 17.2), вызываемое из окна **Properties** нажатием кнопки [...]. В нашем случае в это окно надо ввести три варианта выравнивания, набирая каждый вариант на отдельной строке:

```
По левому краю
По правому краю
По центру
```

Определите обработчик события `CheckedChanged` для компонента `checkBox1` (листинг 23.5).

В описание класса `Form1` добавьте новое поле:

```
private Form2 form2 = new Form2();
```

В конструктор класса `Form1` добавьте оператор

```
AddOwnedForm(form2);
```

Дополните выпадающее меню, связанное с пунктом **Format** *(см. разд. 19.1—19.4)*, добавив в него пункт с текстом `&Paragraph...`. Положите свойство `Name` добавленного пункта меню равным `paragraph1` и определите для этого пункта меню обработчик события `Click` (листинг 23.6).

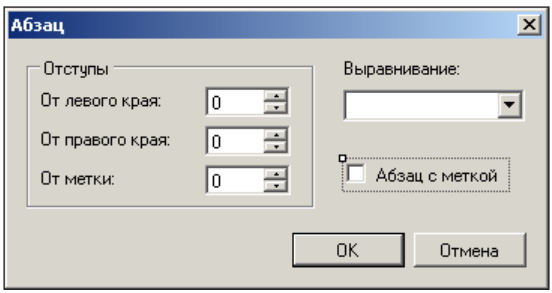


Рис. 23.1. Вид формы `Form2` для проекта TXTEDIT6

Листинг 23.4. Настройка свойств

```
Form2: Text = Абзац, MaximizeBox = False, MinimizeBox = False,
  FormBorderStyle = FixedDialog, StartPosition = CenterScreen,
  ShowInTaskbar = False, AcceptButton = button1, CancelButton = button2
groupBox1: Text = Отступы
label1: Text = От левого края:
label2: Text = От правого края:
```

label3: Text = **От метки:**, Enabled = **False**
label4: Text = **Выравнивание:**
numericUpDown1: Modifiers = **Internal**
numericUpDown2: Modifiers = **Internal**
numericUpDown3: Modifiers = **Internal**, Enabled = **False**
comboBox1: DropDownStyle = **DropDownList**, Modifiers = **Internal**
checkBox1: Text = **Абзац с меткой**, Modifiers = **Internal**
button1: Text = **OK**, DialogResult = **OK**
button2: Text = **Отмена**

Листинг 23.5. Обработчик `checkBox1.CheckedChanged` (форма `Form2`)

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
  label3.Enabled = numericUpDown3.Enabled = checkBox1.Checked;
  numericUpDown3.Value = numericUpDown3.Enabled ? 10 : 0;
}
```

Листинг 23.6. Обработчик `paragraph1.Click` (форма `Form1`)

```
private void paragraph1_Click(object sender, EventArgs e)
{
  form2.checkBox1.Checked = textBox1.SelectionBullet;
  form2.comboBox1.SelectedIndex = (int)textBox1.SelectionAlignment;
  form2.numericUpDown1.Value = textBox1.SelectionIndent;
  form2.numericUpDown2.Value = textBox1.SelectionRightIndent;
  form2.numericUpDown3.Value = textBox1.BulletIndent;
  if (form2.ShowDialog() == DialogResult.OK)
  {
    textBox1.SelectionIndent = (int)form2.numericUpDown1.Value;
    textBox1.SelectionRightIndent = (int)form2.numericUpDown2.Value;
    textBox1.BulletIndent = (int)form2.numericUpDown3.Value;
    textBox1.SelectionBullet = form2.checkBox1.Checked;
    textBox1.SelectionAlignment =
      (HorizontalAlignment)form2.comboBox1.SelectedIndex;
    textBox1_SelectionChanged(this, null);
  }
}
```

Результат. Новая команда меню **Format | Paragraph...** позволяет настраивать свойства текущего абзаца или группы выделенных абзацев. Ее выполнение приводит к появлению диалогового окна **Абзац**, в котором можно указать величину отступов абзаца от левого и правого краев области редактирования (в пикселах) и способ выравнивания. Кроме того, установив флажок **Абзац с меткой**, можно снабдить абзац *меткой* (маркером) в виде черной точки (•); в этом случае можно указать величину отступа от метки до текста.

Комментарии

1. В данном разделе была использована еще одна группа свойств компонента `RichTextBox`, связанных с настройками абзаца: `SelectionIndent` и `SelectionRightIndent` (задают смещение в пикселах соответственно от левого и правого края области редактирования), `SelectionBullet` (логического типа; если равно `true`, то абзац снабжается меткой), `BulletIndent` (задает отступ в пикселах от метки до следующего за ней текста).
2. Порядок пунктов в выпадающем списке `comboBox1` соответствует порядку, в котором различные варианты выравнивания указаны в перечислении `HorizontalAlignment`: `Left` (`0`), `Right` (`1`), `Center` (`2`). Благодаря этому обстоятельству, для определения выбранного варианта выравнивания достаточно преобразовать свойство `SelectedIndex` компонента `comboBox1`, содержащее номер выбранного пункта списка, к типу `HorizontalAlignment` (см. предпоследний оператор в листинге 23.6). Последующий вызов обработчика `textBox1_SelectedChanged` обеспечивает корректировку вида кнопок быстрого доступа и пунктов меню, связанных с выравниванием.

23.4. Отображение текущей строки и столбца

Действуя таким же образом, как в *разд. 22.1*, добавьте на статусную панель `statusStrip1` еще одну (пятую) метку с именем **`position1`** и настройте свойства добавленной метки (листинг 23.7).

Зацепите мышью добавленную метку `position1` и переместите ее левее метки `hint1` (с текстом "Ready"). В результате указанные метки поменяются местами (рис. 23.2).

В *начало* метода `textBox1_SelectionChanged`, приведенного в листинге 23.3, добавьте новый фрагмент (листинг 23.8).

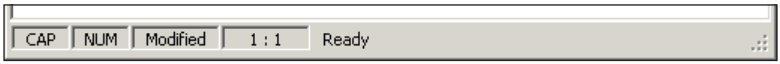


Рис. 23.2. Вид нижней части формы `Form1` для проекта TXTEDIT6

Листинг 23.7. Настройка свойств

```
position1: Text = 1 : 1, AutoSize = False, BorderSides = All,
  BorderStyle = Sunken, Size.Width = 60
```

Листинг 23.8. Добавление в начало метода `textBox1_SelectionChanged`

```
int x = textBox1.SelectionStart,
  y = textBox1.GetLineFromCharIndex(x),
  x0 = textBox1.GetFirstCharIndexFromLine(y);
position1.Text = string.Format("{0} : {1}", y + 1, x - x0 + 1);
```

Результат. На статусной панели отображается текущая позиция клавиатурного курсора *(каретки)* в формате *номер строки* **:** *номер символа в строке* (строки и символы нумеруются от 1). Если редактируемый текст содержит выделенный фрагмент, то указывается позиция начала этого фрагмента.

Комментарий

Для определения текущей позиции клавиатурного курсора в тексте компонента `textBox1` (этот текст содержится в свойстве `Text` данного компонента) использованы свойства и методы, имеющиеся и у компонента `TextBox`, и у компонента `RichTextBox`. Это уже упоминавшееся ранее свойство `SelectionStart` (см. комментарий 2 в *разд. 8.1*), позволяющее определить позицию курсора или начало выделенного фрагмента в строке `Text`, а также методы `GetLineFromCharIndex(n)` (позволяет по номеру `n` символа в строковом свойстве `Text` определить номер строки, в которой данный символ находится) и `GetFirstCharIndexFromLine(n)` (позволяет определить номер символа в свойстве `Text`, с которого начинается строка с номером `n`). И строки, и символы нумеруются от 0, поэтому в листинге 23.8 к полученным значениям прибавляются единицы. Вместо метода `GetFirstCharIndexFromLine(n)` можно использовать метод `GetFirstCharIndexOfCurrentLine` (без параметров), который позволяет определить номер символа в свойстве `Text`, с которого начинается *текущая* строка (т. е. строка, содержащая клавиатурный курсор).

Отметим, что содержимое компонентов `TextBox` и `RichTextBox`, разбитое на строки, можно получить с помощью свойства `Lines` — массива строк. Наконец, упомянем еще один метод, имеющий отношение к позиции клавиатурного

курсора: ScrollToCaret. Этот метод, не имеющий параметров, позволяет отобразить на экране фрагмент текста, содержащий клавиатурный курсор (если в момент вызова метода клавиатурный курсор уже присутствует на экране, то метод ScrollToCaret не выполняет никаких действий).

23.5. Загрузка и сохранение текста без форматных настроек

Для компонентов saveFileDialog1 и openFileDialog1 измените значения свойства Filter на **RTF files|*.rtf|Text files|*.txt|All files|*.***. В результате с указанными диалоговыми окнами будут связаны *три* фильтра: для RTF-файлов (с расширением rtf), обычных текстовых файлов (с расширением txt) и произвольных файлов, имеющих любое допустимое имя и расширение.

В класс Form1 добавьте новый метод GetFileType (листинг 23.9).

Измените методы SaveToFile и open1_Click (листинг 23.10).

Листинг 23.9. Метод GetFileType формы Form1

```
private RichTextBoxStreamType GetFileType(string path)
{
  string s = Path.GetExtension(path).ToUpper();
  return s == ".RTF" ? RichTextBoxStreamType.RichText :
    RichTextBoxStreamType.PlainText;
}
```

Листинг 23.10. Новый вариант методов SaveToFile и open1_Click

```
private void SaveToFile(string path)
{
  textBox1.SaveFile(path, GetFileType(path));
  textBox1.Modified = false;
}
private void open1_Click(object sender, EventArgs e)
{
  if (TextSaved())
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
      string path = openFileDialog1.FileName;
      textBox1.LoadFile(path, GetFileType(path));
```

```
        Text = "Text Editor - " + Path.GetFileName(path);
        saveFileDialog1.FileName = path;
        openFileDialog1.FileName = "";
    }
}
```

Результат. При сохранении документа в файле с любым расширением, отличным от rtf, в файл записывается только текст (без форматных настроек), причем в *ANSI-кодировке*, т. е. в кодировке, используемой системой Windows по умолчанию. Однако в самом редакторе форматные настройки сохраняются, и в дальнейшем текст вместе с форматными настройками можно сохранить в файле с расширением rtf. Загрузить в редактор теперь можно как файлы в формате RTF (с расширением rtf), так и обычные текстовые файлы. Для того чтобы ускорить выбор файлов с расширениями rtf и txt, а также иметь возможность выбора файлов с любым расширением, в окнах открытия и сохранения файлов предусмотрены соответствующие *фильтры* (**RTF files**, **Text files**, **All files**), перечисленные в выпадающем списке **Тип файла**.

Приведем изображение работающей программы (рис. 23.3). Клавиатурный курсор располагается на первой, отцентрированной, строке.

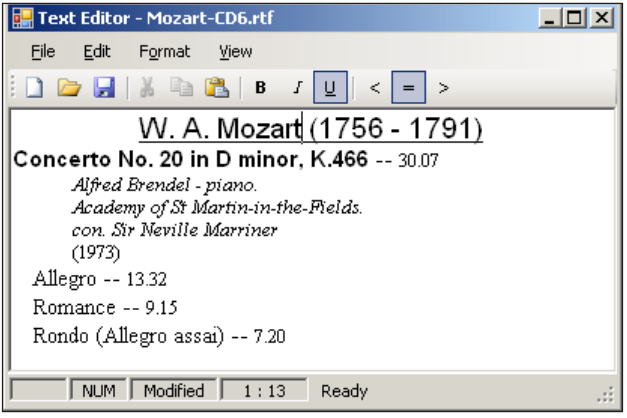


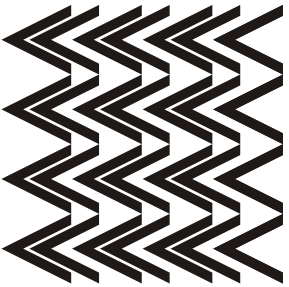
Рис. 23.3. Вид работающего приложения TXTEDIT6

Комментарии

1. Для выделения расширения из имени файла (см. листинг 23.9) был использован метод `GetExtension` класса `Path`, возвращающий расширение файла вместе с предшествующей точкой. Если имя файла не содержит расши-

рения или оканчивается точкой, то возвращается пустая строка. По поводу использования метода `ToUpper` см. комментарий 6 в *разд 14.1*.

2. Методы `SaveFile` и `LoadFile` компонента `RichTextBox` позволяют сохранять и считывать данные в различных форматах. Формат определяется вторым параметром этих методов, имеющим перечислимый тип `RichTextBoxStreamType`. В новом варианте программы используется не только формат `RichText` (для RTF-файлов), но и формат `PlainText`, позволяющий сохранять текст без форматных настроек в стандартной ANSI-кодировке Windows. Среди других форматов отметим `UnicodePlainText`, позволяющий сохранять текст без форматных настроек в кодировке Unicode.
3. При работе с диалоговым окном сохранения файла `saveFileDialog1` пользователь может испытывать определенные неудобства. Например, если он создал файл test.rtf и затем захотел сохранить его в формате TXT, то в окне сохранения ему потребуется стереть расширение rtf в поле ввода **Имя файла** и явным образом ввести расширение txt. В подобных ситуациях более естественным действием представляется изменение типа файла с помощью выпадающего списка **Тип файла**. Однако изменение типа файла в этом списке *не сопровождается* автоматическим изменением файлового расширения в поле **Имя файла**. Требуемое изменение можно было бы выполнять в обработчике события, связанного с выбором нового типа файла, но, к сожалению, в компонентах `SaveFileDialog` и `OpenFileDialog` такое событие не предусмотрено.



Часть III

Глава 24

Флажки и группы флажков: проект CHKBOXES

Проект CHKBOXES посвящен компонентам `CheckBox` и `CheckedListBox`, реализующим элементы управления "флажок" и "группа флажков". Рассматриваются методы, обеспечивающие проверку и изменение состояния флажков, а также особенности использования флажков, принимающих три состояния.

24.1. Установка флажков и проверка их состояния

После создания проекта CHKBOXES разместите в форме `Form1` три флажка (`checkBox1`—`checkBox3`), три метки (`label1`—`label3`) и кнопку `button1`. Настройте свойства формы и добавленных компонентов (листинг 24.1) и расположите компоненты в соответствии с рис. 24.1. Поскольку свойство `Text` для меток `label1`—`label3` имеет одно и то же значение, удобно настроить это свойство сразу для всех меток, предварительно выделив их.

Добавьте к проекту новую форму (она получит имя `Form2`) и разместите в ней три компонента — *списка флажков* типа `CheckedListBox` (они получат имена `checkedListBox1`—`checkedListBox3`) и кнопку `button1`. Настройте свойства формы `Form2` и ее компонентов (листинг 24.2) и расположите компоненты в соответствии с рис. 24.2. При определении свойства `Items` компонентов `CheckedListBox` используется специальное диалоговое окно (рис. 17.2); в нашем случае в это окно надо ввести указанные числа (5 чисел для `checkedListBox1`, 4 числа для `checkedListBox2`, 6 чисел для `checkedListBox3`), набирая каждое число *на отдельной строке*.

В описание класса `Form1` добавьте новое поле:

```
private Form2 form2 = new Form2();
```

В конструктор класса `Form1` добавьте оператор

```
AddOwnedForm(form2);
```

Определите обработчик события `Click` для кнопки `button1` формы `Form1` (листинг 24.3), а также обработчик события `FormClosed` для формы `Form2` (листинг 24.4).

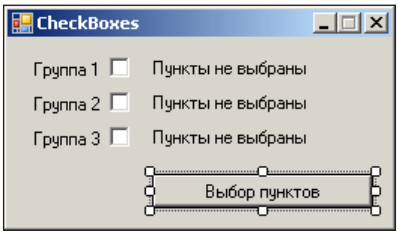


Рис. 24.1. Вид формы `Form1` для проекта CHKBOXES

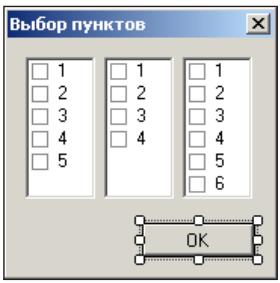


Рис. 24.2. Вид формы `Form2` для проекта CHKBOXES

Листинг 24.1. Настройка свойств формы `Form1` и ее компонентов

```
Form1: Text = CheckBoxes, MaximizeBox = False,
  FormBorderStyle = FixedSingle, AcceptButton = button1
checkBox1: Text = Группа 1, RightToLeft = Yes
checkBox2: Text = Группа 2, RightToLeft = Yes
checkBox3: Text = Группа 3, RightToLeft = Yes
label1–label3: Text = Пункты не выбраны
button1: Text = Выбор пунктов
```

Листинг 24.2. Настройка свойств формы `Form2` и ее компонентов

```
Form2: Text = Выбор пунктов, MaximizeBox = False, MinimizeBox = False,
  FormBorderStyle = FixedDialog, StartPosition = CenterScreen,
  ShowInTaskbar = False, AcceptButton = button1
checkedListBox1: Items = 1 2 3 4 5, CheckOnClick = true
```

checkedListBox2: Items = **1 2 3 4**, CheckOnClick = **true**
checkedListBox3: Items = **1 2 3 4 5 6**, CheckOnClick = **true**
button1: Text = **OK**, DialogResult = **OK**

Листинг 24.3. Обработчик `button1.Click` для формы `Form1`

```
private void button1_Click(object sender, EventArgs e)
{
  form2.ShowDialog();
}
```

Листинг 24.4. Обработчик `Form2.FormClosed`

```
private void Form2_FormClosed(object sender, FormClosedEventArgs e)
{
  for (int i = 1; i <= 3; i++)
  // обработка каждой группы флажков
  {
    string s = "";
    CheckedListBox clb =
      Controls["checkedListBox" + i] as CheckedListBox;
    int k = clb.CheckedIndices.Count;
    if (k == 0)
      s = "Пункты не выбраны";
    else
      if (k == clb.Items.Count)
        s = "Выбраны все пункты";
      else
      {
        foreach (int j in clb.CheckedIndices)
          s = s + " " + clb.Items[j].ToString();
        s = "Выбрано:" + s;
      }
    // вывод информации о флажках в соответствующей метке формы Form1
    Owner.Controls["label" + i].Text = s;
  }
}
```

Результат. Если вызвать диалоговое окно **Выбор пунктов**, установить в нем какие-либо флажки и закрыть его, то в главной форме **CheckBoxes** отобразится информация о выбранных пунктах для каждой группы флажков. Флажки главной формы пока не используются.

Комментарии

1. Для того чтобы установить флажок (или снять установленный флажок) в списке флажков `CheckedListBox`, можно выделить данный флажок с помощью клавиш со стрелками и нажать клавишу <Пробел>. Способ действия, использующий мышь, зависит от значения свойства `CheckOnClick`. Если это свойство равно **`False`** (значение по умолчанию), то выделение флажка щелчком мыши еще не приводит к изменению его состояния; для изменения состояния флажка необходимо еще раз щелкнуть мышью на уже выделенном флажке. Если же свойство `CheckOnClick` имеет значение **`True`**, то выделение флажка щелчком мыши сразу приводит к изменению его состояния. Второй способ представляется более естественным, поскольку именно таким образом ведет себя по умолчанию одиночный флажок (компонент `CheckBox`).

2. Контролировать действия пользователя, связанные с перемещением по списку флажков и изменением их состояния, можно с помощью обработчиков двух событий.

   Событие `SelectedIndexChanged` происходит при изменении выделенного элемента с помощью клавиатуры или мыши, а также при щелчке мышью на уже выделенном элементе (определить индекс выделенного флажка можно с помощью свойства `SelectedIndex`, доступного как для чтения, так и для записи; индексация, как обычно, ведется от нуля).

   Событие `ItemCheck` происходит при попытке изменения состояния флажка. С помощью свойств второго параметра `e` обработчика этого события можно определить предыдущее состояние (свойство `CurrentValue`), новое состояние (`NewValue`) и индекс флажка (`Index`). Свойства `CurrentValue` и `NewValue` имеют перечислимый тип `CheckState` (с тремя возможными значениями: `Checked`, `Unchecked` и `Indeterminate`). Важно отметить, что свойство `NewValue` в обработчике *можно изменять*; эта возможность особенно полезна при использовании списков флажков с тремя состояниями, так как с помощью действий пользователя нельзя установить флажок в состояние `Indeterminate` (с этим обстоятельством связано также отсутствие у компонента `CheckedListBox` свойства `ThreeState`, имеющегося у одиночного флажка `CheckBox` и позволяющего пользователю последовательно устанавливать одиночный флажок в три состояния).

3. Получить информацию об установленных флажках проще всего с помощью свойства-коллекции `CheckedIndices`, содержащего индексы установленных флажков (под установленными понимаются флажки в состояниях `Checked` и `Indeterminate`). Количество установленных флажков можно определить с помощью свойства `Count` коллекции `CheckedIndices` (разумеется, свойство `Count` есть и у коллекции `Items`, содержащей *все* флажки).

   Для получения текста, связанного с элементом коллекции `CheckedIndices` или `Items`, следует вызвать для этого элемента метод `ToString`.

   Если требуется уточнить состояние флажка, то можно воспользоваться методом `GetItemCheckState(ind)` компонента `CheckedListBox`, позволяющим определить состояние флажка с индексом `ind` (метод возвращает значение перечислимого типа `CheckState`). Парный к нему метод `SetItemCheckState(ind, state)` позволяет установить флажок с индексом `ind` в состояние `state`. Если состояние `Indeterminate` не используется, то проще применять методы `GetItemChecked(ind)` (возвращает значение логического типа) и `SetItemChecked(ind, value)` с параметром `value` логического типа. В этих методах предполагается, что значение `true` соответствует установленному флажку, а значение `false` — снятому.

4. Для перебора индексов всех установленных флажков мы использовали цикл `foreach` (см. листинг 24.4) как более удобный в указанной ситуации (перебор элементов коллекции без их изменения). Однако можно было использовать и цикл `for`, так как для доступа к элементам любой коллекции можно применять операцию индексирования.

24.2. Глобальная установка флажков

Определите обработчик события `CheckStateChanged` для флажка `checkBox1` (листинг 24.5), после чего свяжите созданный обработчик с событием `CheckStateChanged` флажков `checkBox2` и `checkBox3`.

Кроме того, измените метод `button1_Click` формы `Form1` (листинг 24.6).

Листинг 24.5. Обработчик `checkBox1.CheckStateChanged`

```
private void checkBox1_CheckStateChanged(object sender, EventArgs e)
{
  CheckBox cb = sender as CheckBox;
  string s = "label" + cb.Name[cb.Name.Length - 1];
  Controls[s].Text = cb.Checked ? "Выбраны все пункты" :
    "Пункты не выбраны";
}
```

Листинг 24.6. Новый вариант метода button1_Click формы Form1

```
private void button1_Click(object sender, EventArgs e)
{
  for (int i = 1; i <= 3; i++)
  {
    CheckedListBox clb =
      form2.Controls["checkedListBox" + i] as CheckedListBox;
    bool b = (Controls["checkBox" + i] as CheckBox).Checked;
    for (int j = 0; j < clb.Items.Count; j++)
      clb.SetItemChecked(j, b);
  }
  form2.ShowDialog();
}
```

Результат. Установка флажка в главной форме обеспечивает выбор всех пунктов соответствующей группы, а его снятие обеспечивает отмену всех выбранных пунктов. При открытии диалогового окна **Выбор пунктов** его списки флажков корректируются.

Недочет. Явно сделанные установки в диалоговом окне не влияют на вид флажков главной формы. Заметим, что если в диалоговом окне выбрана часть пунктов, то соответствующий флажок главной формы желательно изображать затененным (т. е. находящимся в состоянии `Indeterminate`). Необходимые исправления будут сделаны в следующем разделе.

> ***ПРИМЕЧАНИЕ***
>
> Вместо события `CheckStateChanged` глобальную установку флажков можно было бы связать с событием `CheckedChanged`. Нами было выбрано событие `CheckStateChanged`, поскольку оно (в отличие от события `CheckedChanged`) возникает и при переходе флажка из выбранного состояния в затененное и обратно. Эта особенность события `CheckStateChanged` нам пригодится в дальнейшем.

24.3. Использование флажков, принимающих три состояния

Измените метод `Form2_FormClosed` формы `Form2` (листинг 24.7).

Листинг 24.7. Новый вариант метода Form2_FormClosed формы Form2

```
private void Form2_FormClosed(object sender, FormClosedEventArgs e)
{
  for (int i = 1; i <= 3; i++)
  {
    CheckedListBox clb =
      Controls["checkedListBox" + i] as CheckedListBox;
    CheckBox cb = Owner.Controls["checkBox" + i] as CheckBox;
    int k = clb.CheckedIndices.Count;
    if (k == 0)
      cb.Checked = false;
    else
      if (k == clb.Items.Count)
        cb.CheckState = CheckState.Checked;
      else
      {
        string s = "";
        foreach (int j in clb.CheckedIndices)
          s = s + " " + clb.Items[j].ToString();
        s = "Выбрано:" + s;
        Owner.Controls["label" + i].Text = s;
        cb.CheckState = CheckState.Indeterminate;
      }
  }
}
```

Результат. В случае выбора части флажков в одном из списков окна **Выбор пунктов** соответствующий флажок в окне **CheckBoxes** становится затененным. Заметим, что пользователь не может явным образом устанавливать флажки окна **CheckBoxes** в затененное состояние, поскольку для компонентов `checkBox1`—`checkBox3` свойство `ThreeState` равно **`False`** (значение по умолчанию).

ПРИМЕЧАНИЕ

Обратите внимание на то, что в новом варианте обработчика `Form2_FormClosed` текст метки изменяется только при выборе *части пунктов*. Если выбраны все пункты или не выбрано ни одного, то изменение текста метки происходит в обработчике `checkBox1_CheckStateChanged` (см. листинг 24.5), который автоматически вызывается при любом изменении состояния флажка.

Ошибка 1. Рядом с затененным флажком может отображаться текст **Выбраны все пункты** (это произойдет в ситуации, когда состояние флажка изменяется с установленного или снятого на затененное). Ошибка связана с тем, что при подобном изменении свойства `CheckState` флажка автоматически вызывается обработчик события `CheckStateChanged`, причем в этом случае свойство `Checked` имеет значение `true`.

Исправление. Откорректируйте метод `checkBox1_CheckStateChanged` класса `Form1` (листинг 24.8).

Листинг 24.8. Новый вариант метода `checkBox1_CheckStateChanged` формы `Form1`

```
private void checkBox1_CheckStateChanged(object sender, EventArgs e)
{
  CheckBox cb = sender as CheckBox;
  if (cb.CheckState == CheckState.Indeterminate)
    return;
  string s = "label" + cb.Name[cb.Name.Length - 1];
  Controls[s].Text = cb.Checked ? "Выбраны все пункты" :
    "Пункты не выбраны";
}
```

Ошибка 2. При повторном вызове окна **Выбор пунктов** затененные флажки, подобно установленным, приводят к установке *всех* флажков в соответствующем списке (эта ошибка, как и ошибка 1, связана с тем, что свойство `Checked` имеет значение `true` как для установленных, так и для затененных флажков).

Исправление. Откорректируйте метод `button1_Click` формы `Form1` (листинг 24.9).

Листинг 24.9. Новый вариант метода `button1_Click` формы `Form1`

```
private void button1_Click(object sender, EventArgs e)
{
  for (int i = 1; i <= 3; i++)
  {
    CheckedListBox clb =
      form2.Controls["checkedListBox" + i] as CheckedListBox;
    CheckState cs = (Controls["checkBox" + i] as CheckBox).CheckState;
    if (cs == CheckState.Indeterminate)
```

```
      continue;
    for (int j = 0; j < clb.Items.Count; j++)
      clb.SetItemCheckState(j, cs);
  }
  form2.ShowDialog();
}
```

Результат. Теперь в случае затененных флажков связанные с ними списки флажков при открытии окна **Выбор пунктов** не изменяются. Для этого в новом варианте метода `button1_Click` используется оператор `continue`, который обеспечивает немедленное завершение текущей итерации цикла.

Глава 25

Выпадающие и обычные списки: проект LISTBOX1

Проект LISTBOX1 посвящен компонентам `ComboBox` и `ListBox`, реализующим элементы управления "выпадающий список" и "список". Рассматриваются методы, обеспечивающие выполнение стандартных действий над элементами списка (добавление, удаление, вставка, изменение порядка следования и т. д.), и описываются действия, позволяющие реализовать перетаскивание элементов списка в режиме drag & drop. В проекте также рассматривается перечисление `KnownColor` и метод `Beep` класса `Console`.

25.1. Создание и использование выпадающих списков

После создания проекта LISTBOX1 разместите в форме `Form1` выпадающий список `comboBox1` и панель `panel1` (рис. 25.1). Настройте свойства формы и добавленных компонентов (листинг 25.1). Отметим, что, положив свойство `DropDownStyle` компонента `comboBox1` равным **`DropDownList`**, мы тем самым отключили возможность ввода новых значений для этого компонента (разрешено выбирать только значения из имеющегося списка).

Дополните конструктор класса `Form1` (листинг 25.2) и определите обработчик события `SelectedIndexChanged` для компонента `comboBox1` (листинг 25.3).

Листинг 25.1. Настройка свойств

```
Form1: Text = ComboBox and ListBox, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
comboBox1: DropDownStyle = DropDownList
panel1: BorderStyle = Fixed3D
```

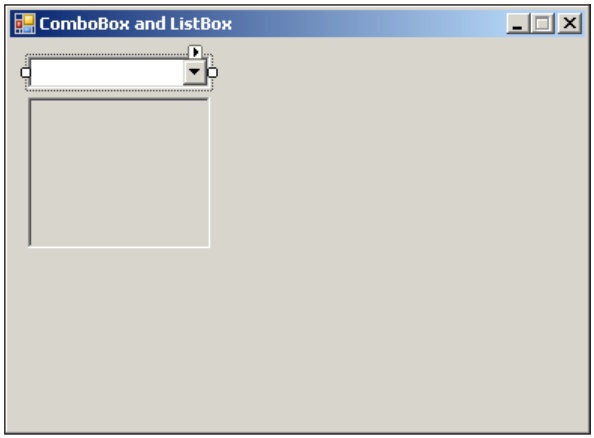


Рис. 25.1. Вид формы `Form1` для проекта LISTBOX1 на начальном этапе разработки

Листинг 25.2. Новый вариант конструктора формы `Form1`

```
public Form1()
{
  InitializeComponent();
  string[] s = Enum.GetNames(typeof(KnownColor));
  int n1 = Array.IndexOf(s, "AliceBlue"),
    cnt = Array.IndexOf(s, "YellowGreen") - n1 + 1;
  string[] s0 = new string[cnt];
  Array.Copy(s,n1,s0,0,cnt);
  comboBox1.Items.AddRange(s0);
  comboBox1.SelectedIndex = 0;
}
```

Листинг 25.3. Обработчик `comboBox1.SelectedIndexChanged`

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  panel1.BackColor =
    Color.FromName(comboBox1.Items[comboBox1.SelectedIndex].ToString());
}
```

Результат. При запуске программы в выпадающем списке содержатся значения идентификаторов для всех именованных цветов. Идентификаторы указаны в алфавитном порядке. Выбор идентификатора из списка приводит к соответствующему изменению цвета панели `panel1`.

Комментарии

1. При формировании списка всех цветов, имеющих имена, было использовано перечисление `KnownColor`. В этом перечислении вначале идут имена *системных* цветов (т. е. цветов, связанных с различными оконными элементами Windows; эта группа состоит из 26 имен), затем идет цвет `Transparent` (который определяется как прозрачный белый цвет), а затем располагаются имена конкретных цветов в алфавитном порядке, начиная с `AliceBlue` и заканчивая `YellowGreen` (количество таких цветов равно 140). Заметим, что в версии .NET 2.0 в конец перечисления `KnownColor` было добавлено еще 7 системных цветов.

   Получить массив всех имен, входящих в перечисление, можно с помощью метода `GetNames` класса `Enum`. Для выделения из полученного массива диапазона, содержащего только имена конкретных (не системных) цветов, используется метод `Copy` класса `Array`. Индексы цветов `AliceBlue` и `YellowGreen` определяются с помощью метода `IndexOf` класса `Array` (можно было использовать и константы `28` и `167`, но это сделало бы текст программы менее понятным).

2. Для добавления очередного элемента в выпадающий список компонента `ComboBox` предназначен метод `Add` свойства-коллекции `Items` этого компонента. Однако, если требуется сразу добавить в список *все* элементы из некоторого массива строк (как в нашем случае), удобнее и быстрее воспользоваться методом `AddRange`.

3. Следует учитывать тот факт, что элементы коллекции `Items` описаны как `object`, поэтому для получения их строкового представления необходимо использовать метод `ToString` (см. листинг 25.3).

25.2. Список: добавление и удаление элементов

Разместите в форме `Form1` список `listBox1`, панель `panel2`, две метки `label1` и `label2` (компонент `panel2` следует разместить под компонентом `panel1`, как показано на рис. 25.2). Настройте свойства добавленных компонентов (листинг 25.4).

Определите обработчики события SelectedIndexChanged для компонента listBox1 и события Click для компонентов label1 и label2 (листинг 25.5).

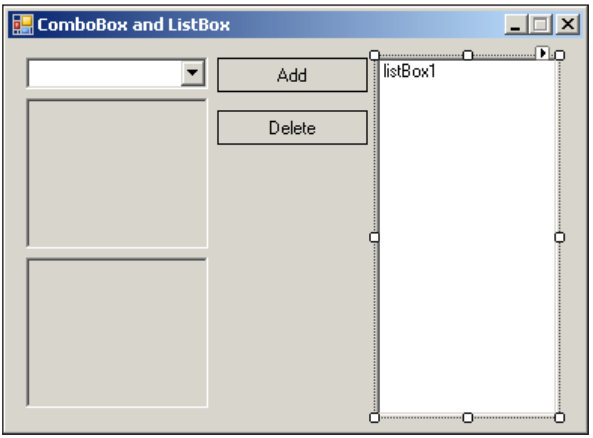


Рис. 25.2. Вид формы Form1 для проекта LISTBOX1 на промежуточном этапе разработки

Листинг 25.4. Настройка свойств

```
panel2: BorderStyle = Fixed3D, Visible = False
label1: Text = Add, AutoSize = False, TextAlign = MiddleCenter,
  BorderStyle = FixedSingle
label2: Text = Delete, AutoSize = False, TextAlign = MiddleCenter,
  BorderStyle = FixedSingle
```

Листинг 25.5. Обработчики listBox1.SelectedIndexChanged, label1.Click и label2.Click

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  panel2.Visible = listBox1.Items.Count > 0;
  if (listBox1.SelectedIndex == -1)
    return;
  panel2.BackColor =
    Color.FromName(listBox1.Items[listBox1.SelectedIndex].ToString());
}
private void label1_Click(object sender, EventArgs e)
```

```
{
  listBox1.SelectedIndex = listBox1.Items.Add(comboBox1.Text);
}
private void label2_Click(object sender, EventArgs e)
{
  listBox1.Items.RemoveAt(listBox1.SelectedIndex);
}
```

Результат. При щелчке мышью на метке **Add** в список `listBox1` добавляется строка, указанная в поле выпадающего списка `comboBox1`. Цвет панели `panel2` соответствует выделенному элементу списка `listBox1`. Если этот список является пустым, то панель `panel2` не отображается. При щелчке на метке **Delete** из списка удаляется выделенный элемент. См. также комментарии 1 и 2.

Недочет. После выполнения операции удаления в списке `listBox1` исчезает выделенный элемент (хотя элемент, обведенный рамкой, остается). В этой ситуации свойство `SelectedIndex` компонента `listBox1` равно `-1`, поэтому повторный щелчок на метке **Delete** приводит к ошибке времени выполнения `ArgumentOutOfRangeException` (так как значение `-1` не является допустимым значением для метода `RemoveAt`). По этой же причине к ошибке приводит щелчок на метке **Delete** в случае пустого списка.

Исправление. Измените метод `label2_Click` (листинг 25.6).

Листинг 25.6. Новый вариант метода `label2_Click`

```
private void label2_Click(object sender, EventArgs e)
{
  int i = listBox1.SelectedIndex;
  if (i == -1)
  {
    Console.Beep();
    return;
  }
  listBox1.Items.RemoveAt(i);
  if (i == listBox1.Items.Count)
    i--;
  listBox1.SelectedIndex = i;
}
```

Результат. При удалении элемента в середине списка `listBox1` выделение сохраняется на текущей позиции (которую теперь занимает следующий элемент). При удалении элемента в конце списка выделяется предыдущий элемент. Таким образом, если список не пуст, он всегда имеет выделенный элемент. При щелчке на метке **Delete** в случае пустого списка выдается звуковой сигнал (см. комментарий 3).

> ***ПРИМЕЧАНИЕ***
>
> Обратите внимание на то, что в методе `listBox1_SelectedIndexChanged` мы предусмотрели особую обработку ситуации, когда свойство `SelectedIndex` имеет значение `-1` (см. листинг 25.6), поскольку такая ситуация возникает в процессе удаления элемента списка.

Комментарии

1. *Выделенный* элемент списка изображается на цветном (обычно синем) фоне. Если список имеет фокус (т. е. является активным компонентом формы) и при этом хотя бы один раз для переключения между компонентами формы использовалась клавиша <Tab>, то выделенный элемент дополнительно обводится пунктирной рамкой. Элемент, обведенный такой рамкой, называется *текущим*. Если свойство списка `SelectionMode` имеет значение **`MultiSimple`** или **`MultiExtended`**, то в списке может находиться несколько выделенных элементов, однако текущий элемент всегда один (и он не обязательно будет выделенным). Если свойство `SelectionMode` имеет значение **`One`**, то текущий и выделенный элементы совпадают (однако при *программных* изменениях списка возможно их рассогласование — см. описание недочета в данном разделе).
2. Для того чтобы следить за сменой текущего (обведенного рамкой) элемента списка при выполнении различных операций, необходимо, чтобы список не терял фокуса. Поэтому выполнение операций мы связали с метками `Label`: эти компоненты, в отличие от компонентов-кнопок `Button`, не могут получать фокус. Таким образом, в форме располагаются лишь два компонента, которые могут получать фокус: это `comboBox1` и `listBox1`. Переключаясь между ними с помощью клавиши <Tab>, легко проследить за особенностями отображения списка, имеющего и не имеющего фокус.
3. Помимо метода `Beep` без параметров класс `Console` имеет вариант данного метода с двумя параметрами типа `int`, первый из которых определяет частоту звука (в диапазоне от 37 до 32 767 Гц), а второй — продолжительность звучания (в миллисекундах).

25.3. Дополнительные операции над списком

Разместите в форме Form1 пять новых меток (label3—label7) и настройте их свойства (листинг 25.7; поскольку свойства AutoSize, TextAlign и BorderStyle для всех меток имеют одинаковые значения, удобно установить их одновременно, предварительно выделив все метки). Расположите новые метки в соответствии с рис. 25.3.

В начало файла Form1.cs добавьте оператор

```
using System.IO;
```

Определите обработчики события Click для добавленных меток (листинг 25.8).

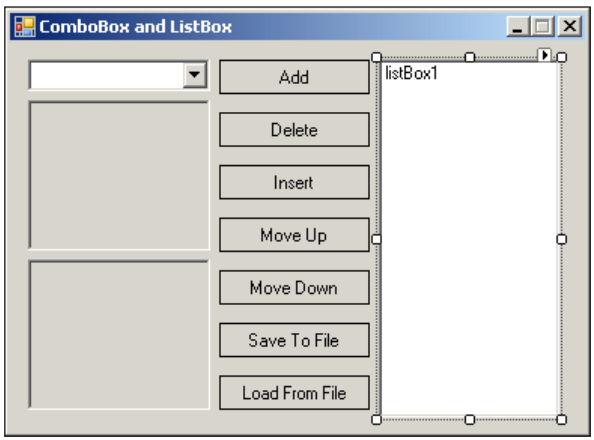


Рис. 25.3. Окончательный вид формы Form1 для проекта LISTBOX1

Листинг 25.7. Настройка свойств

```
label3: Text = Insert, AutoSize = False, TextAlign = MiddleCenter,
  BorderStyle = FixedSingle
label4: Text = Move Up, AutoSize = False, TextAlign = MiddleCenter,
  BorderStyle = FixedSingle
label5: Text = Move Down, AutoSize = False, TextAlign = MiddleCenter,
  BorderStyle = FixedSingle
label6: Text = Save To File, AutoSize = False, TextAlign = MiddleCenter,
  BorderStyle = FixedSingle
label7: Text = Load From File, AutoSize = False, TextAlign =
  MiddleCenter, BorderStyle = FixedSingle
```

Листинг 25.8. Обработчики `label3.Click`, `label4.Click`, `label5.Click`, `label6.Click` и `label7.Click`

```
private void label3_Click(object sender, EventArgs e)
{
  int i = listBox1.SelectedIndex;
  if (i == -1)
    label1_Click(this, null);
  else
  {
    listBox1.Items.Insert(i, comboBox1.Text);
    listBox1.SelectedIndex = i;
  }
}
private void label4_Click(object sender, EventArgs e)
{
  int i = listBox1.SelectedIndex;
  if (i <= 0)
  {
    Console.Beep();
    return;
  }
  object x = listBox1.Items[i];
  listBox1.Items[i] = listBox1.Items[i - 1];
  listBox1.Items[i - 1] = x;
  listBox1.SelectedIndex = i - 1;
}
private void label5_Click(object sender, EventArgs e)
{
  int i = listBox1.SelectedIndex;
  if (i == listBox1.Items.Count - 1)
  {
    Console.Beep();
    return;
  }
  object x = listBox1.Items[i];
  listBox1.Items[i] = listBox1.Items[i + 1];
```

```
  listBox1.Items[i + 1] = x;
  listBox1.SelectedIndex = i + 1;
}
private void label6_Click(object sender, EventArgs e)
{
  if (listBox1.Items.Count == 0)
  {
    Console.Beep();
    return;
  }
  StreamWriter sw = new StreamWriter("LISTBOX1.dat");
  foreach (object o in listBox1.Items)
    sw.WriteLine(o);
  sw.Close();
}
private void label7_Click(object sender, EventArgs e)
{
  if (!File.Exists("LISTBOX1.dat"))
  {
    Console.Beep();
    return;
  }
  listBox1.Items.Clear();
  StreamReader sr = new StreamReader("LISTBOX1.dat");
  while (!sr.EndOfStream)
    listBox1.Items.Add(sr.ReadLine());
  sr.Close();
  listBox1.SelectedIndex = listBox1.Items.Count - 1;
}
```

Результат. Щелчок мышью на метке **Insert** вставляет новый элемент перед выделенным и выделяет вставленный элемент (в случае пустого списка команда **Insert** и команда **Add** выполняют одинаковые действия). Команды **Move Up** и **Move Down** позволяют перемещать выделенный элемент соответственно вверх и вниз по списку, сохраняя его выделение (при попытке выполнить перемещение первого элемента вверх или последнего элемента вниз выдается

звуковой сигнал). Команда **Save To File** позволяет сохранить содержимое *непустого* списка в файле LISTBOX1.dat, а команда **Load From File** — загрузить в список данные из этого файла (если файл отсутствует, то при попытке загрузить данные выдается звуковой сигнал).

Комментарии

1. При реализации новых действий были использованы методы `Insert` (вставка нового элемента в указанную позицию) и `Clear` (очистка списка элементов) коллекции `Items` компонента `ListBox`.
2. Для проверки существования файла мы воспользовались методом `Exists` класса `File`. Файл LISTBOX1.dat содержит только латинские буквы, поэтому при создании потоков для его записи и чтения допустимо не указывать применяемый формат кодирования, так как в подобной ситуации форматы UTF-8 и ANSI совпадают (по поводу форматов кодирования см. комментарий 2 в *разд. 18.2*). Обратите внимание на использование свойства `EndOfStream` класса `StreamReader` для проверки того, достигнут ли конец текстового файла (это свойство появилось в .NET 2.0).

25.4. Выполнение операций над списком с помощью мыши

Свяжите событие `DoubleClick` компонента `listBox1` с существующим обработчиком `label2_Click` (его текст приведен в листинге 25.6).

Положите свойство `AllowDrop` компонента `listBox1` равным **`True`** и добавьте в класс `Form1` поле, которое в режиме перетаскивания будет содержать индекс перетаскиваемого элемента (source) из списка `listBox1`:

```
private int iSrc;
```

Определите обработчики события `MouseDown` для компонентов `comboBox1` и `listBox1`, а также обработчики событий `DragEnter` и `DragDrop` для компонента `listBox1` (листинг 25.9).

Листинг 25.9. Обработчики `comboBox1.MouseDown`, `listBox1.MouseDown`, `listBox1.DragEnter` и `listBox1.DragDrop`

```
private void comboBox1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Right)
```

```
    DoDragDrop(comboBox1.Text, DragDropEffects.Copy);
}
private void listBox1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Right)
  {
    iSrc = listBox1.IndexFromPoint(e.Location);
    if (iSrc != ListBox.NoMatches)
      DoDragDrop(listBox1.Items[iSrc].ToString(), DragDropEffects.Move);
  }
}
private void listBox1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.All;
}
private void listBox1_DragDrop(object sender, DragEventArgs e)
{
  if (e.AllowedEffect == DragDropEffects.Move)
    listBox1.Items.RemoveAt(iSrc);
  string s = e.Data.GetData(typeof(string)) as string;
  int iTrg =
    listBox1.IndexFromPoint(listBox1.PointToClient(new Point(e.X, e.Y)));
  if (iTrg == ListBox.NoMatches)
    listBox1.SelectedIndex = listBox1.Items.Add(s);
  else
  {
    listBox1.Items.Insert(iTrg, s);
    listBox1.SelectedIndex = iTrg;
  }
}
```

Результат. При двойном щелчке мышью на элементе списка `listBox1` происходит удаление этого элемента (благодаря связыванию события `DoubleClick` компонента `listBox1` с обработчиком `label2_Click`). Кроме того, теперь для перемещения элемента списка на новую позицию можно использовать механизм drag & drop: достаточно зацепить любой (не обязательно выделенный) элемент *правой* кнопкой мыши и перетащить его на новое место. Текст, ото-

бражаемый в поле выпадающего списка comboBox1, также можно поместить в список listBox1 с помощью перетаскивания правой кнопкой мыши. Если текст перетаскивается на существующий элемент списка, то он вставляется в указанную позицию, а если текст перетаскивается на свободную область списка, то он добавляется к списку. В любом случае он становится выделенным элементом списка.

При перетаскивании текста из выпадающего списка comboBox1 изображение курсора содержит символ "+", что является признаком режима перетаскивания **Copy** (Копировать); при перетаскивании элемента списка listBox1 на новое место курсор не содержит символа "+", что является признаком режима **Move** (Переместить). См. также комментарии 1—3.

Недочет. В начале перетаскивания текста из выпадающего списка comboBox1 курсор имеет вид запрещающего знака.

Исправление. Положите свойство AllowDrop компонента comboBox1 равным **True** и определите обработчик события DragEnter для компонента comboBox1 (листинг 25.10).

Листинг 25.10. Обработчик comboBox1.DragEnter

```
private void comboBox1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Copy;
}
```

Результат. Теперь при перетаскивании текста из выпадающего списка comboBox1 курсор над этим списком является разрешающим (хотя, разумеется, отпускание перетаскиваемого элемента над выпадающим списком не приведет ни к каким результатам). Отметим, что если перетаскивание начато из обычного списка listBox1, то над компонентом comboBox1 курсор будет иметь запрещающий вид (это связано с тем, что выпадающий список в качестве приемника перетаскивания реагирует только на режим **Copy**).

Комментарии

1. Выбор *правой* кнопки мыши в качестве инициатора режима перетаскивания связан с тем, что и выпадающий список comboBox1, и обычный список listBox1 должны стандартным образом реагировать на левую кнопку мыши: выпадающий список при щелчке левой кнопкой разворачивается, а в обычном списке выделяется тот элемент, на котором произведен щелчок. Кроме того, с двойным щелчком на списке listBox1 мы также связали особое действие

(удаление элемента). Если бы режим перетаскивания активизировался по нажатию левой кнопки, то стандартные действия, связанные с левой кнопкой, было бы невозможно выполнить.

2. Методы, события и свойства, связанные с перетаскиванием, были подробно описаны в *главе 11*. Единственной особенностью реализации режима drag & drop, приведенной в листинге 25.9, является применение метода `IndexFromPoint(p)` компонента `ListBox`, который позволяет определить индекс элемента списка, содержащего точку `p` с указанными координатами (если в указанной точке не содержится элемент списка, то метод возвращает особое значение `ListBox.NoMatches`). Метод `IndexFromPoint` используется в обработчиках `listBox1_MouseDown` и `listBox1_DragDrop`. Так как в этом методе требуется указывать локальные координаты точки `p` относительно компонента `ListBox`, а в обработчиках событий, связанных с перетаскиванием, можно получить только экранные координаты, в обработчике `listBox1_DragDrop` при определении индекса `iTrg` элемента-приемника (target) приходится дополнительно использовать метод `listBox1.PointToClient` (в обработчике `listBox1_MouseDown` вызывать данный метод не требуется, так как в обработчиках событий, связанных с мышью, возвращаются локальные координаты).

3. Благодаря использованию *разных* режимов перетаскивания (**Copy** и **Move**), по виду курсора можно определить, какой компонент является источником данных (для режима `DragDropEffects.Copy` это `comboBox1`, для режима `DragDropEffects.Move` — `listBox1`). С помощью проверки текущего режима перетаскивания (а именно свойства `e.AllowedEffect`) в начале метода `listBox1_DragDrop` определяется, надо ли удалять элемент-источник из списка (впрочем, здесь можно было обойтись и без использования указанного свойства; для этого достаточно в режиме перетаскивания из выпадающего списка `comboBox1` присвоить полю `iSrc` какое-либо особое значение, например, `-1`, а в начале метода `listBox1_DragDrop` проверить значение этого поля).

Глава 26

Списки с множественным выделением и графические списки: проект LISTBOX2

Проект LISTBOX2 посвящен дополнительным возможностям компонента `ListBox`, реализующего элемент управления "список". Рассматриваются различные варианты списков с множественным выделением и особенности выполнения для них стандартных действий над элементами списка. Кроме того, описываются свойства и события компонента `ListBox`, обеспечивающие графический режим представления элементов списка.

26.1. Списки с множественным выделением

В качестве заготовки для проекта LISTBOX2 следует использовать ранее разработанный проект LISTBOX1 *(см. главу 25)*. Скопируйте проект LISTBOX1 в новый каталог LISTBOX2 и выполните действия, необходимые для переименования проекта *(см. разд. 1.1)*.

Разместите в форме `Form1` компонент-контейнер `groupBox1`. В компоненте `groupBox1` разместите три радиокнопки (`radioButton1`—`radioButton3`). Настройте свойства добавленных компонентов (листинг 26.1; рис. 26.1).

Определите обработчик события `CheckedChanged` для радиокнопки `radioButton1` (листинг 26.2), после чего свяжите созданный обработчик с событием `CheckedChanged` радиокнопок `radioButton2` и `radioButton3`.

Листинг 26.1. Настройка свойств

```
groupBox1: Text = SelectionMode
radioButton1: Text = One, Checked = True
radioButton2: Text = MultiSimple
radioButton3: Text = MultiExtended
```

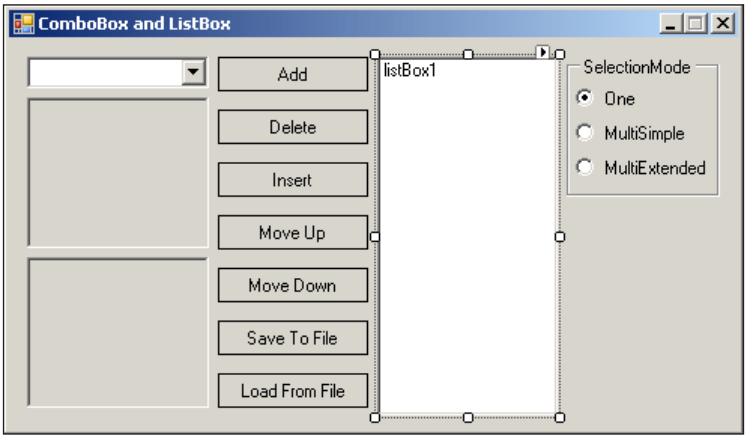


Рис. 26.1. Вид формы `Form1` для проекта LISTBOX2 на начальном этапе разработки

Листинг 26.2. Обработчик `radioButton1.CheckedChanged`

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
  RadioButton rb = sender as RadioButton;
  if (!rb.Checked)
    return;
  listBox1.SelectionMode = (SelectionMode)(rb.TabIndex + 1);
  ActiveControl = listBox1;
}
```

Результат. С помощью набора радиокнопок можно устанавливать значение свойства `SelectionMode`, задающего режим выделения элементов списка:

- `One` — режим, не допускающий множественного выделения. Этот режим был рассмотрен в предыдущем примере *(см. главу 25)*;
- `MultiSimple` — режим, допускающий множественное выделение, но не позволяющий автоматически выделять наборы смежных элементов;
- `MultiExtended` — режим, допускающий множественное выделение, в том числе и автоматическое выделение смежных элементов.

В режиме `MultiSimple` для выделения элемента достаточно щелкнуть на нем мышью или, сделав его текущим (т. е. подведя к нему с помощью клавиш со стрелками пунктирную рамку), нажать клавишу <Пробел>. Если элемент уже был выделен, то описанные действия снимают выделение.

В режиме `MultiExtended` реализованы приемы выделения, аналогичные тем, которые используются в системе Windows:

- при щелчке мышью на элементе он становится выделенным (и одновременно текущим), выделение с других элементов снимается;
- при щелчке с нажатой клавишей <Ctrl> элемент становится текущим и его выделение меняется на противоположное (если элемент не был выделен, то он выделяется, если был выделен, то выделение с него снимается); выделение для прочих элементов сохраняется;
- при щелчке с нажатой клавишей <Shift> выделяются все элементы, расположенные между текущим элементом и элементом, на котором произведен щелчок;
- при перемещении мыши с нажатой левой кнопкой по элементам списка производится выделение всех этих элементов.

Для выделения с помощью клавиатуры режим `MultiExtended` предоставляет существенно меньше возможностей по сравнению с режимом `MultiSimple`: так, с помощью клавиатуры нельзя выделить несколько *несмежных* элементов. Для выделения смежных элементов надо сделать текущим первый (или последний) из них и затем перемещать рамку текущего элемента с помощью клавиш со стрелками, держа нажатой клавишу <Shift>.

В режимах `MultiSimple` и `MultiExtended` работают все ранее определенные для списка операции, однако следует учитывать, что теперь они выполняются над *первым* выделенным элементом. См. также комментарии 1 и 2.

Недочет 1. При переключении из режима множественного выделения в режим **`One`** остается выделенным *последний* из ранее выделенных элементов, однако на панели `panel2` сохраняется цвет, соответствующий *первому* из ранее выделенных элементов.

Исправление. Добавьте в конец метода `radioButton1_CheckedChanged` оператор

```
listBox1_SelectedIndexChanged(this, null);
```

Недочет 2. Команды **Move Up** и **Move Down** в режиме множественного выделения работают не вполне корректно: если, например, выделены два смежных элемента **Black** и **Blue** (выше находится **Black**), то первый вызов команды **Move Down** переместит **Black** ниже **Blue**, но второй вызов *восстановит первоначальный порядок* (так как после перемещения элемента **Black** вниз первым выделенным элементом станет **Blue**, и именно на него будет распространяться действие очередной команды). В случае *одного* выделенного элемента его перемещение вверх в режиме множественного выделения приведет к тому, что за ним будет оставаться "след" из выделенных элементов, а при перемещении

вниз ситуация будет подобна той, которая была описана ранее для двух выделенных элементов. Эти особенности выполнения команд объясняются тем, что в режиме множественного выделения при выделении нового элемента не происходит автоматического снятия выделения с остальных элементов.

Исправление. В начало методов `label4_Click` и `label5_Click`, описанных в листинге 25.8, добавьте следующий фрагмент:

```
if (listBox1.SelectedIndices.Count > 1)
{
  Console.Beep();
  return;
}
```

В конец этих методов добавьте оператор

```
listBox1.SetSelected(i, false);
```

Результат. Теперь команды **Move Up** и **Move Down** выполняются только в случае, если в списке имеется единственный выделенный элемент; при этом в любом режиме они работают корректным образом (см. комментарий 3). Если попытаться выполнить одну из указанных команд при наличии в списке нескольких выделенных элементов, то будет выдан звуковой сигнал.

Комментарии

1. При установке режима выделения в методе `radioButton1_Click` (см. листинг 26.2) было использовано то обстоятельство, что элементы `One`, `MultiSimple` и `MultiExtended` перечисления `SelectionMode` имеют значения `1`, `2` и `3` соответственно, а свойства `TabIndex` радиокнопок, расположенных в компоненте `groupBox1`, равны `0`, `1` и `2`. Перечисление `SelectionMode` имеет также элемент `None` со значением `0`; он соответствует режиму, в котором список не может содержать ни одного выделенного элемента. Поскольку при этом свойство `SelectedIndex` всегда равно `–1`, трудно представить ситуацию, в которой подобный режим выделения может оказаться полезным.
2. Последний оператор метода `radioButton1_CheckedChanged` в листинге 26.2 обеспечивает немедленное перемещение фокуса к списку `listBox1` после изменения режима выделения элементов списка.
3. При исправлении второго недочета мы воспользовались следующими методами и свойствами компонента `ListBox`, предназначенными для режима множественного выделения:
   - метод `SetSelected(ind, value)` позволяет установить (если `value` равно `true`) или снять выделение (если `value` равно `false`) для

элемента с индексом ind; парным к данному методу является метод GetSelected(ind) логического типа, позволяющий определить, выделен ли элемент списка с индексом ind;

- свойство-коллекция SelectedIndices содержит индексы всех выделенных элементов списка (количество выделенных элементов можно получить с помощью свойства Count данной коллекции); отметим, что свойство SelectedIndex всегда содержит индекс *первого* из выделенных элементов.

26.2. Обработка выделенных элементов

Разместите в форме Form1 две новые метки (label8 и label9) и настройте их свойства (листинг 26.3; поскольку свойства AutoSize, TextAlign и BorderStyle для меток имеют одинаковые значения, удобно установить их одновременно, предварительно выделив обе метки). Разместите метки в соответствии с рис. 26.2.

Определите обработчик события Click для метки label8 (листинг 26.4), после чего свяжите созданный обработчик с событием Click метки label9.

Кроме того, измените метод label2_Click (листинг 26.5).

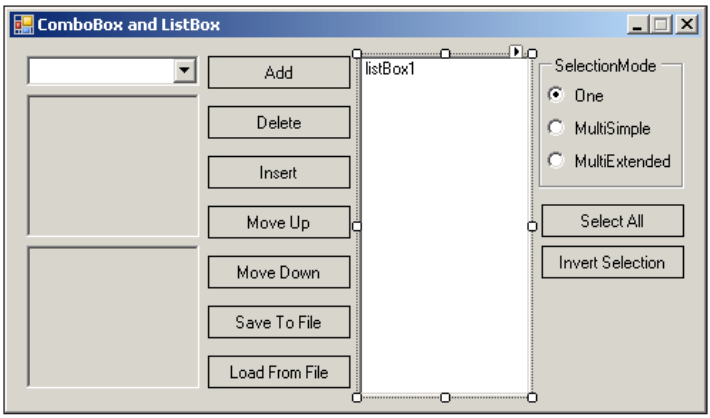


Рис. 26.2. Вид формы Form1 для проекта LISTBOX2 на промежуточном этапе разработки

Листинг 26.3. Настройка свойств

```
label8: Text = Select All, AutoSize = False, TextAlign = MiddleCenter,
  BorderStyle = FixedSingle
label9: Text = Invert Selection, AutoSize = False,
  TextAlign = MiddleCenter, BorderStyle = FixedSingle
```

Листинг 26.4. Обработчик label8.Click

```
private void label8_Click(object sender, EventArgs e)
{
  if (listBox1.SelectionMode == SelectionMode.One)
  {
    Console.Beep();
    return;
  }
  for (int i = 0; i < listBox1.Items.Count; i++)
    if (sender == label8)
      listBox1.SetSelected(i, true);
    else
      listBox1.SetSelected(i, !listBox1.GetSelected(i));
  listBox1_SelectedIndexChanged(this, null);
}
```

Листинг 26.5. Новый вариант метода label2_Click

```
private void label2_Click(object sender, EventArgs e)
{
  int i = listBox1.SelectedIndex;
  if (i == -1)
  {
    Console.Beep();
    return;
  }
  if (listBox1.SelectionMode == SelectionMode.One)
  {
    listBox1.Items.RemoveAt(i);
    if (i == listBox1.Items.Count)
      i--;
    listBox1.SelectedIndex = i;
  }
  else
    for (int j = listBox1.SelectedIndices.Count - 1; j >= 0; j--)
      listBox1.Items.RemoveAt(listBox1.SelectedIndices[j]);
}
```

Результат. В режимах `MultiSimple` и `MultiExtended` щелчок мышью на метке **Select All** приводит к выделению всех элементов списка, а щелчок на метке **Invert Selection** *инвертирует* выделение: выделенные элементы становятся невыделенными и наоборот. В режиме `One` щелчок на этих метках приводит к звуковому сигналу.

Кроме того, в режимах `MultiSimple` и `MultiExtended` выполнение команды **Delete** теперь приводит к удалению *всех* выделенных элементов.

> ***ПРИМЕЧАНИЕ***
>
> Поскольку удаление любого элемента списка приводит к изменению коллекции `SelectedIndices`, для корректного удаления всех выделенных элементов необходимо перебирать элементы коллекции `SelectedIndices` *с конца*, что и было сделано в новом варианте метода `label2_Click` (см. листинг 26.5).

Недочет. В режиме множественного выделения возможна ситуация, когда непустой список не содержит ни одного выделенного элемента (эта ситуация может возникнуть, например, при удалении всех выделенных элементов или при последовательном выполнении команд **Select All** и **Invert Selection**). Однако в такой ситуации панель `panel2` остается окрашенной в цвет, соответствующий ранее выделенному элементу. Кроме того, если в ситуации, когда список в режиме множественного выделения не содержит ни одного выделенного элемента, перейти в режим `One`, то в списке по-прежнему не будет выделен ни один элемент, что нарушает правило об *обязательном* выделении одного элемента в режиме `One` для непустых списков.

Исправление. Измените начальную часть метода `listBox1_SelectedIndexChanged` (листинг 26.6).

Листинг 26.6. Новый вариант метода `listBox1_SelectedIndexChanged`

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  panel2.Visible = listBox1.Items.Count > 0 &&
    (listBox1.SelectionMode == SelectionMode.One ||
    listBox1.SelectedIndices.Count > 0);
  if (panel2.Visible && listBox1.SelectedIndex == -1)
    listBox1.SelectedIndex = 0;
  if (listBox1.SelectedIndex == -1)
    return;
  panel2.BackColor =
    Color.FromName(listBox1.Items[listBox1.SelectedIndex].ToString());
}
```

Результат. Теперь панель `panel2` делается невидимой не только в случае пустого списка, но и в случае списка, не содержащего ни одного выделенного элемента. Условие `listBox1.SelectionMode == SelectionMode.One` было добавлено для того, чтобы не допустить "мерцания" панели `panel2` при удалении элемента в режиме `One` (поскольку в тот момент, когда выделенный элемент удален, а следующий или предыдущий еще не выделен, свойство `SelectedIndices.Count` равно `0`). Далее, благодаря добавленному оператору `if`, при переходе в режим `One` в случае, если ранее список не содержал выделенных элементов, выделяется *начальный* элемент списка.

Комментарии

1. При использовании режима множественного выделения надо учитывать, что в этом режиме свойство `SelectedIndex` ведет себя достаточно странным образом. Как уже было сказано, оно всегда возвращает индекс первого выделенного элемента. Однако ему можно присвоить индекс любого другого элемента, при этом указанный элемент будет выделен. Однако если этот элемент находится *ниже* первого выделенного, то при чтении свойства `SelectedIndex` мы опять получим индекс первого выделенного элемента (иными словами, не то значение, которое было ранее присвоено этому свойству). Эта особенность затрудняет обработку события `SelectedIndexChanged`: данное событие может наступить при выделении второго элемента списка, но мы не сможем этого узнать, если ранее был выделен первый элемент, поскольку именно его индекс будет возвращен свойством `SelectedIndex`. Всех отмеченных несообразностей можно было избежать, если бы в режиме множественного выделения свойство `SelectedIndex` связывалось с *текущим* элементом; при этом для обработки выделенных элементов можно было бы использовать другие методы и свойства (см., в частности, методы и свойства, перечисленные в комментарии 3 в *разд. 26.1*).

2. К сожалению, класс `ListBox` не имеет средств для управления *текущим* (т. е. обведенным пунктирной рамкой) элементом списка в режиме множественного выделения: нельзя ни узнать, какой элемент является текущим, ни программно переместить пунктирную рамку на другой элемент. Более того, даже при наличии у списка фокуса ввода рамка выделения может отсутствовать, что делает практически невозможной работу со списком в режиме `MultiSimple` с помощью клавиатуры (в *разд. 25.2* было отмечено, что для появления рамки вокруг текущего элемента необходимо хотя бы один раз воспользоваться клавишей <Tab>).

   Поэтому если в программе желательно иметь возможность управления текущим элементом в режиме множественного выделения, вместо компо-

нента ListBox удобнее использовать компонент CheckedListBox *(см. главу 24)*, выделяя его элементы с помощью установки связанных с ними флажков.

26.3. Графические списки

Разместите в форме Form1 еще один компонент-контейнер типа GroupBox (он получит имя groupBox2). В компоненте groupBox2 разместите две радиокнопки (radioButton4 и radioButton5). Настройте свойства добавленных компонентов (листинг 26.7; рис. 26.3).

Определите обработчик события CheckedChanged для радиокнопки radioButton4 (листинг 26.8), после чего свяжите созданный обработчик с событием CheckedChanged радиокнопки radioButton5.

В класс Form1 добавьте поле

```
private SolidBrush brush = new SolidBrush(Color.Black);
```

и определите обработчик события DrawItem для списка listBox1 (листинг 26.9).

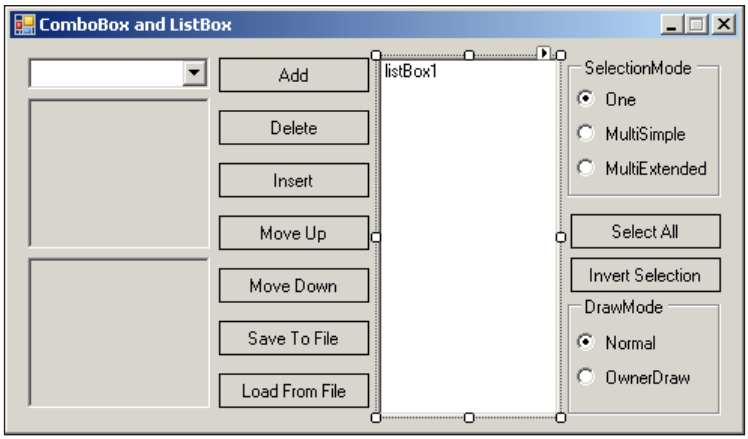


Рис. 26.3. Окончательный вид формы Form1 для проекта LISTBOX2

Листинг 26.7. Настройка свойств

```
groupBox1: Text = DrawMode
radioButton4: Text = Normal, Checked = True
radioButton5: Text = OwnerDraw
```

Листинг 26.8. Обработчик `radioButton4.CheckedChanged`

```
private void radioButton4_CheckedChanged(object sender, EventArgs e)
{
  RadioButton rb = sender as RadioButton;
  if (!rb.Checked)
    return;
  listBox1.DrawMode = (DrawMode)rb.TabIndex;
  ActiveControl = listBox1;
}
```

Листинг 26.9. Обработчик `listBox1.DrawItem`

```
private void listBox1_DrawItem(object sender, DrawItemEventArgs e)
{
  if (e.Index == -1)
    return;
  brush.Color = Color.FromName(listBox1.Items[e.Index].ToString());
  e.Graphics.FillRectangle(brush, e.Bounds);
}
```

Результат. При выборе в наборе радиокнопок **DrawMode** варианта **OwnerDraw** в списке `listBox1` отображаются не текстовые строки, а прямоугольные области соответствующего цвета. Выбор варианта **Normal** восстанавливает исходное (текстовое) представление списка. См. также комментарии 1 и 2.

Недочет. В графическом режиме нельзя определить, какой элемент является текущим, а какие — выделенными.

Исправление. В метод `listBox1_DrawItem` добавьте новый фрагмент (листинг 26.10).

Листинг 26.10. Добавление к методу `listBox1_DrawItem`

```
Rectangle r = e.Bounds;
r.Width--; r.Height--;
if ((e.State & DrawItemState.Focus) == DrawItemState.Focus)
  e.Graphics.DrawRectangle(Pens.Black, r);
if ((e.State & DrawItemState.Selected) == DrawItemState.Selected)
```

```
{
  r.Height++;
  r.Width /= 10;
  brush.Color = Color.Black;
  e.Graphics.FillRectangle(brush, r);
}
```

Результат. В режиме `OwnerDraw` вокруг текущего элемента рисуется черная рамка толщиной в 1 пиксел. Кроме того, выделенные элементы отмечаются слева небольшой черной полосой (ширина этой полосы равна 1/10 ширины элемента списка). См. также комментарии 3 и 4.

Комментарии

1. Способ отображения элементов списка в графических стилях `OwnerDrawFixed` и `OwnerDrawVariable` определяется в обработчике события `DrawItem`, параметр `e` которого содержит свойство `Index` (порядковый номер элемента, нумерация от 0) и `Bounds` (прямоугольная область, занимаемая этим элементом на экране). Для рисования следует использовать объект типа `Graphics`, который также предоставляется параметром `e` в виде одноименного свойства. Стиль `OwnerDrawVariable`, в отличие от стиля `OwnerDrawFixed`, позволяет изображать элементы списка *разного размера*. При использовании стиля `OwnerDrawVariable` размер любого элемента можно задать в обработчике события `MeasureItem` (это событие возникает непосредственно перед событием `DrawItem`).
2. В случае пустого графического списка обработчик `listBox1_DrawItem` вызывается с параметром `e.Index`, равным `-1`. Условный оператор, указанный в обработчике (см. листинг 26.9), необходим для того, чтобы в подобной ситуации не было возбуждено исключение при попытке обращения к элементу коллекции `Items` с индексом `-1`.
3. Для исправления отмеченного недочета (см. листинг 26.10) было использовано свойство `State` параметра `e`, имеющее перечислимый тип `DrawItemState` и содержащее информацию о состоянии изображаемого элемента. Если `State` содержит значение `DrawItemState.Focus`, значит, изображаемый элемент является текущим. Если `State` содержит значение `DrawItemState.Selected`, значит, этот элемент является выделенным. Свойство `State` может содержать одновременно *оба* эти значения (а также более 10 других значений, предусмотренных в перечислении `DrawItemState`), поэтому для проверки наличия требуемого значения приходится применять операцию `&` (побитовое И).

Следует заметить, что у параметра e обработчика события `DrawItem` имеются также два метода (без параметров): `DrawBackground`, который закрашивает фон элемента списка, и `DrawFocusRectangle`, который рисует пунктирную рамку вокруг текущего элемента. При закрашивании фона учитывается, является ли элемент выделенным. Цвет, используемый при закрашивании фона, содержится в свойстве `e.BackColor`, которое доступно только для чтения. Имеется также свойство `e.ForeColor`, определяющее стандартный цвет рисуемого элемента (для выделенных и невыделенных элементов это свойство, как и свойство `BackColor`, принимает *различные* значения). Указанные методы и свойство `ForeColor` были использованы в проекте PNGEDIT4 при реализации выпадающего списка стилей для рисования линий (*см. разд. 17.5*, листинг 17.10).

4. Уменьшение на 1 пиксел ширины и высоты прямоугольной области рисования (см. вторую строку в листинге 26.10) требуется для того, чтобы компенсировать "ошибку смещения на 1 пиксел", возникающую при рисовании контуров прямоугольников и эллипсов (см. описание недочета в *разд. 16.3*). Если этого не сделать, то при смене текущего элемента списка у прежнего текущего элемента будет сохраняться часть черной рамки. Поскольку при заливке ошибка смещения отсутствует, далее в листинге 26.10 восстанавливается исходное значение высоты `Height` области рисования.

Глава 27

MDI-приложение: проект JPEGVIEW

Проект JPEGVIEW посвящен *приложениям с многодокументным интерфейсом* (MDI-приложениям). Рассматриваются вопросы, связанные со взаимодействием главной формы и дочерних форм (child-форм), описываются способы реализации стандартных действий над дочерними формами (варианты размещения на экране, вывод списка child-форм в меню, одновременное закрытие всех child-форм и т. д.). Кроме того, реализуется режим масштабирования изображения в child-форме и возможность прокрутки содержимого формы с помощью клавиатуры.

27.1. Открытие и закрытие дочерних форм в MDI-приложении

После создания проекта JPEGVIEW добавьте в форму `Form1` невизуальные компоненты типа `MenuStrip` и `OpenFileDialog` (эти компоненты получат имена `menuStrip1` и `openFileDialog1`; они будут размещены в области невизуальных компонентов под изображением формы, кроме того, в верхней части формы появится меню, связанное с компонентом `menuStrip1`).

Используя конструктор меню *(см. разд. 18.1)*, создайте в компоненте `menuStrip1` пункт меню первого уровня с текстом **`&File`** и с помощью окна **Properties** измените имя этого пункта (т. е. свойство `Name`) на **`file1`**. В выпадающем меню, связанном с пунктом **File**, создайте пункт с текстом **`&Open`**, затем пункт с текстом **`-`** (дефис; этот пункт будет преобразован в горизонтальную разделительную линию), затем пункт **`E&xit`**. Настройте свойства формы `Form1` и добавленных в нее компонентов и пунктов меню (листинг 27.1; рис. 27.1).

Добавьте к проекту новую форму (она получит имя `Form2`) и разместите в ней компонент-меню `menuStrip1` и компонент-контейнер для изображений `pictureBox1`. В компоненте `menuStrip1` формы `Form2` создайте пункт меню первого уровня с текстом **`&File`** и именем **`file1`**. В выпадающем меню, связанном с пунктом **File**, создайте два пункта меню с текстом **`&Open`** и **`&Close`**, затем

пункт с текстом – (дефис), затем пункт **E&xit**. Настройте свойства формы Form2 и добавленных в нее компонентов и пунктов меню (листинг 27.2; рис. 27.2).

ПРИМЕЧАНИЕ

Поскольку компонент pictureBox1 будет заслонять строку меню (см. рис. 27.2), выбрать пункт меню **File** (а значит, и развернуть выпадающее меню, связанное с этим пунктом) в форме Form2 не удастся. В этой ситуации для выбора пунктов меню можно воспользоваться выпадающим списком в верхней части окна **Properties**.

Определите обработчики события Click для пунктов меню open1 и exit1 формы Form1 (листинг 27.3).

Определите обработчики события Click для пунктов меню open1, close1 и exit1 формы Form2 (листинг 27.4).

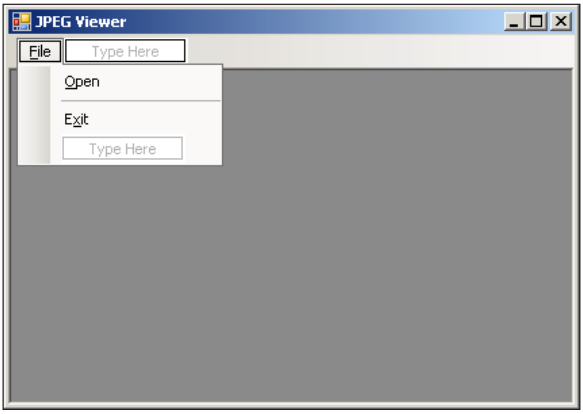


Рис. 27.1. Вид формы Form1 для проекта JPEGVIEW на начальном этапе разработки

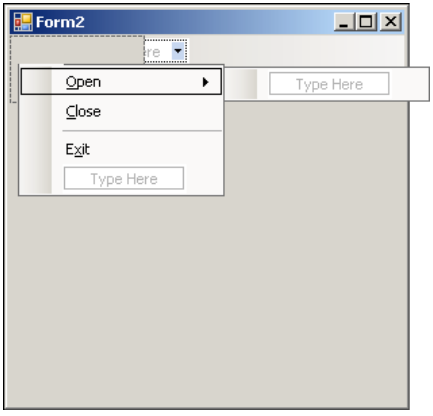


Рис. 27.2. Вид формы Form2 для проекта JPEGVIEW на начальном этапе разработки

Листинг 27.1. Настройка свойств формы Form1 и ее компонентов

```
Form1: Text = JPEG Viewer, IsMdiContainer = True,
  StartPosition = CenterScreen
openFileDialog1: Title = Открытие файла,
  Filter = JPEG Images|*.jpg;*.jpeg
пункт меню Open (группа File): Name = open1
пункт меню EXit (группа File): Name = exit1
```

Листинг 27.2. Настройка свойств формы Form2 и ее компонентов

```
Form2: AutoScroll = True
pictureBox1: SizeMode = AutoSize, Location = 0; 0, Modifiers = Internal
file1: MergeAction = Replace
пункт меню Open (группа File): Name = open1
пункт меню Close (группа File): Name = close1
пункт меню EXit (группа File): Name = exit1
```

Листинг 27.3. Обработчики open1.Click и exit1.Click (для пунктов меню формы Form1)

```
internal void open1_Click(object sender, EventArgs e)
// модификатор доступа изменен на internal
{
  openFileDialog1.FileName = "";
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    // создание child-формы и ее настройка:
    Form2 f = new Form2();
    string n = openFileDialog1.FileName;
    Image i = Image.FromFile(n);
    f.pictureBox1.Image = i;
    f.Text = string.Format("{0} ({1}x{2})",
      System.IO.Path.GetFileNameWithoutExtension(n), i.Width, i.Height);
    f.MdiParent = this;
    f.Show();
  }
```

```
}
private void exit1_Click(object sender, EventArgs e)
{
  Close();
}
```

Листинг 27.4. Обработчики `open1.Click`, `close1.Click` и `exit1.Click` (для пунктов меню формы `Form2`)

```
private void open1_Click(object sender, EventArgs e)
{
  (MdiParent as Form1).open1_Click(this, null);
}
private void close1_Click(object sender, EventArgs e)
{
  // закрытие дочерней формы:
  Close();
}
private void exit1_Click(object sender, EventArgs e)
{
  // закрытие главной формы:
  MdiParent.Close();
}
```

Результат. Из главной формы **JPEG Viewer** с помощью команды **Open** можно загружать графические файлы в формате JPEG (jpg-файлы). Каждый из загруженных файлов отображается в отдельном окне (*дочерней форме*, child-форме), причем в заголовке child-формы указывается имя файла и размер изображения в пикселах (например, **200x350**). Начальный размер и положение child-формы определяются по умолчанию (так как ее свойство `StartPosition` равно **`WindowsDefaultLocation`**). Если размеры child-формы недостаточны для отображения всего изображения, то в ней появляются полосы прокрутки. При разворачивании child-формы ее заголовок совмещается с заголовком главной формы. При сворачивании child-формы ее заголовок помещается в нижней части главной формы. Если открыта хотя бы одна child-форма, то в меню главной формы появляется дополнительная команда **Close**, которая позволяет закрыть активную child-форму. Если закрыты все child-формы, то восстанавливается исходное меню главной формы.

Для закрытия активной child-формы можно использовать комбинацию клавиш <Ctrl>+<F4>. Кроме того, предусмотрены комбинации для активизации следующей и предыдущей child-формы: это, соответственно, <Ctrl>+<F6> и <Ctrl>+<Shift>+<F6>. Переключаться между открытыми child-формами можно также с помощью клавиш со стрелками. Обработка всех упомянутых клавиш производится в MDI-приложении автоматически. См. также комментарии 1—4.

Недочет. Если ширина дочерней формы превосходит размеры рисунка, то на верхней части дочерней формы видна полоса меню.

Исправление. Для компонента `menuStrip1` формы `Form2` положите значение свойства `Visible` равным **`False`**.

Результат. Теперь меню на дочерних формах не отображается, причем это не влияет на вид пунктов, добавленных из этого меню в меню главной формы.

Комментарии

1. Child-форма меню не имеет. При активизации child-формы содержимое ее компонента `menuStrip1` *добавляется* к пунктам меню первого уровня главной формы. Способ добавления определяется свойством `MergeAction` пункта меню дочерней формы: в нашем случае оно равно **`Replace`**, поэтому пункт меню **File** главной формы *заменяется* на соответствующий пункт дочерней формы. Следует заметить, что для возможности объединения меню у компонентов `menuStrip1` как главной, так и дочерней формы свойство `AllowMerge` должно иметь значение **`True`** (это значение по умолчанию).
2. Для доступа к главной форме предусмотрено свойство `MdiParent` дочерней формы. Любая форма с непустым значением свойства `MdiParent` считается дочерней формой. Заметим, что кнопки, соответствующие дочерним формам, на панели задач не отображаются, поэтому устанавливать значение **`False`** для свойства `ShowInTaskbar` дочерних форм не требуется. Для того чтобы определить форму как главную форму MDI-приложения, достаточно положить ее свойство `MdiContainer` равным **`True`**.
3. Для выделения из полного имени файла собственно имени (без пути и расширения) предназначен метод `GetFileNameWithoutExtension` класса `Path` из пространства имен `System.IO`. Поскольку при последующей разработке проекта классы из этого пространства имен использоваться не будут, мы не стали добавлять в начало файла Form1.cs оператор `using System.IO`, а просто указали полное имя класса `System.IO.Path` при обращении к его методу (см. листинг 27.3).

4. Так как для компонента `openFileDialog1` свойство `InitialDirectory` не было задано, при первом запуске программы в качестве начального каталога будет установлен ее рабочий каталог (в нашем случае подкаталог bin\Debug каталога, содержащего проект JPEGVIEW). Если же с помощью диалогового окна `openFileDialog1` будет выбран файл из другого каталога, то при последующем открытии диалогового окна в качестве начального будет предложен именно этот каталог. Более того, информация о последнем использованном каталоге будет автоматически сохранена в специальном разделе реестра Windows и впоследствии этот каталог будет предложен в качестве начального при новом запуске программы. Подобное поведение программы является вполне разумным и не требует от программиста никаких усилий по его реализации — ср. с явным заданием свойства `InitialDirectory` в проектах PNGEDIT1 *(см. разд. 14.1)* и TXTEDIT1 *(см. разд. 18.2 и 18.3)*.

27.2. Стандартные действия над дочерними формами

Создайте в компоненте-меню `menuStrip1` формы `Form1` новый пункт меню первого уровня с текстом **`&Window`** и с помощью окна **Properties** измените имя этого пункта (т. е. свойство `Name`) на **`window1`**. В выпадающем меню, связанном с пунктом **Window**, создайте пункты с текстом **`&HTile`**, **`&VTile`**, **`&Cascade`**, **`&Arrange Icons`** и настройте свойства `Name` этих пунктов, положив их равными **`hTile1`**, **`vTile1`**, **`cascade1`**, **`arrangeIcons1`** соответственно (рис. 27.3).

Добавьте новые операторы в конструктор класса `Form1` (листинг 27.5). Определите обработчик события `Click` для пункта меню `hTile1` (листинг 27.6), после чего свяжите созданный обработчик с событием `Click` пунктов меню `vTile1`, `cascade1` и `arrangeIcons1`.

Листинг 27.5. Новый вариант конструктора формы `Form1`

```
public Form1()
{
  InitializeComponent();
  hTile1.Tag = MdiLayout.TileHorizontal;
  vTile1.Tag = MdiLayout.TileVertical;
  cascade1.Tag = MdiLayout.Cascade;
  arrangeIcons1.Tag = MdiLayout.ArrangeIcons;
}
```

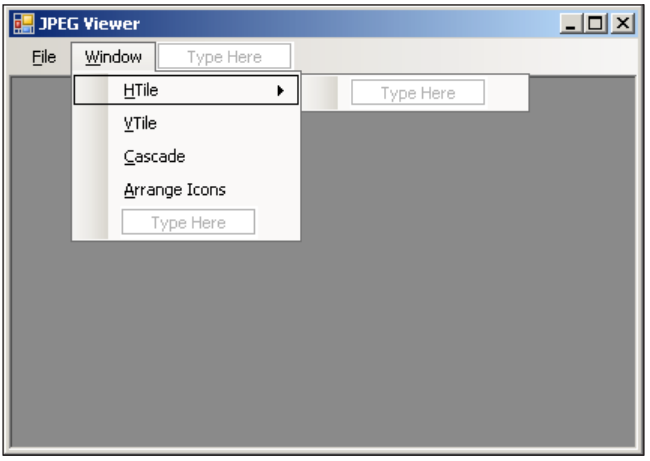


Рис. 27.3. Вид формы `Form1` для проекта JPEGVIEW на промежуточном этапе разработки

Листинг 27.6. Обработчик `hTile1.Click`

```
private void hTile1_Click(object sender, EventArgs e)
{
  LayoutMdi((MdiLayout)(sender as ToolStripMenuItem).Tag);
}
```

Результат. С помощью команд меню группы **Window** можно выполнять стандартные действия над child-формами.

- **HTile** и **VTile**: размещение всех child-форм без перекрытий *(черепицей)* так, чтобы они покрывали всю главную форму. Данные команды различаются способом расположения форм: **HTile** размещает формы сверху вниз (растягивая их по горизонтали), а **VTile** — слева направо (растягивая их по вертикали). Иногда эти команды действуют одинаково (например, для четырех или пяти child-форм).
- **Cascade**: размещение всех child-форм *каскадом* так, чтобы заголовки всех форм были видны на экране. При этом устанавливается единый стандартный размер для всех child-форм, зависящий от текущего размера главной формы.
- **Arrange Icons**: размещение всех *свернутых* child-форм на нижней границе главной формы.

См. также комментарии 1 и 2.

Недочет. При отсутствии child-форм команды группы **Window** теряют смысл, однако по-прежнему доступны в меню.

Исправление. Для пункта меню `window1` формы `Form1` установите значение свойства `Modifiers` равным **`Internal`**, а значение свойства `Visible` равным **`False`**.

В методе `open1_Click` формы `Form1`, описанном в листинге 27.3, перед оператором

```
f.Show();
```

вставьте оператор

```
window1.Visible = true;
```

Определите обработчик события `FormClosed` для формы `Form2` (листинг 27.7).

Листинг 27.7. Обработчик `Form2.FormClosed`

```
private void Form2_FormClosed(object sender, FormClosedEventArgs e)
{
  if (MdiParent.MdiChildren.Length == 1)
    (MdiParent as Form1).window1.Visible = false;
}
```

Результат. Пункт меню **Window** отображается только при наличии хотя бы одной открытой child-формы. См. также комментарии 3 и 4.

Комментарии

1. Все стандартные действия над child-формами реализованы в одном методе `LayoutMdi` класса `Form`; требуемое действие определяется его единственным параметром перечислимого типа `MdiLayout`. Для возможности использования одного общего обработчика `hTile1_Click` для всех команд группы **Window** мы поместили в свойство `Tag` каждой команды соответствующий элемент перечисления `MdiLayout` (листинг 27.5).
2. Обратите внимание на то, что в обработчике `hTile1_Click` (листинг 27.6) дважды выполняется преобразование к новому типу: первый раз для параметра `sender` и второй раз — для свойства `Tag`. При этом в первом случае используется операция `as`, а в другом — операция вида `(`*`тип`*`)`*`выражение`*. Операцию `as` можно применять только для преобразования *ссылочных* типов данных (например, компонентов); для типов-перечислений и числовых

типов данных эту операцию применять нельзя. Операцию вида (*тип*)*выражение* можно использовать для преобразования любых типов данных, однако для *компонентов* ее использовать менее удобно, чем операцию as. Это связано с тем, что операции приведения типа имеют меньший приоритет, чем операция . (точка) доступа к свойству объекта. Например, выражение (ToolStripMenuItem)sender.Tag является синтаксически ошибочным, так как означает преобразование к типу ToolStripMenuItem не объекта sender, а его свойства sender.Tag (ошибка связана с тем, что объект sender имеет тип object, а у типа object нет свойства Tag). Вариант ((ToolStripMenuItem)sender).Tag является правильным, но уступает в наглядности варианту (sender as ToolStripMenuItem).Tag. Скорость выполнения указанных выражений будет практически одной и той же. Напомним, что при применении этих операций к ссылочным данным единственным отличием операции (*тип*)*выражение* от операции as является то, что при невозможности указанного преобразования операция (*тип*) возбуждает исключение InvalidCastException, тогда как операция as просто возвращает значение null.

3. В методе Form2_FormClosed (см. листинг 27.7) использовано свойство-массив MdiChildren типа Form[], содержащее все открытые в текущий момент child-формы. В данном случае нас интересовало только *количество* этих форм, которое можно получить с помощью свойства Length массива MdiChildren: если при выполнении метода Form2_FormClosed оказывается истинным равенство MdiChildren.Length == 1, т. е. в приложении остается *единственная* child-форма (которая в данный момент закрывается), то в меню главной формы надо скрыть пункт **Windows**.
4. При первом обращении к объекту MdiParent в методе Form2_FormClosed (см. листинг 27.7) не требуется преобразовывать его к типу Form1, поскольку свойство MdiChildren имеется уже у класса Form, т. е. у *любой* формы. При обращении к пункту меню window1, который является полем класса Form1 (но не класса Form), преобразование MdiParent к фактическому типу Form1 является обязательным. Заметим также, что для возможности доступа к полю window1 формы Form1 из другого класса (а именно формы Form2) нам потребовалось изменить его свойство Modifiers на **Internal**.

27.3. Добавление в меню списка открытых дочерних форм

Для компонента menuStrip1 формы Form1 установите значение свойства MdiWindowListItem равным **window1**.

Результат. В выпадающем меню **Window** теперь выводится список всех открытых child-форм, причем около активной child-формы рисуется "галочка". Щелкнув на элементе из этого списка, можно активизировать соответствующую child-форму. Для выбора элемента из списка можно также нажать клавишу, соответствующую его порядковому номеру (от <1> до <9>). Если открыто более 9 окон, то ниже пункта, соответствующего девятому окну, выводится пункт **More Windows...**, выбрав который можно отобразить на экране вспомогательное диалоговое окно **Select Window** со списком всех дочерних форм.

Комментарий

Свойство `MdiWindowListItem` позволяет указать, в каком из выпадающих меню главной формы будет выведен список открытых дочерних форм. Это свойство появилось в версии .NET 2.0.

Следует отметить некоторую небрежность в реализации диалогового окна **Select Window**. Так, при изменении его размера по вертикали можно полностью скрыть клиентскую часть данного окна, а при изменении размера по горизонтали — скрыть кнопку **OK**. Далее, кнопка отмены в этом окне содержит английский текст **Cancel**, в то время как в стандартных диалоговых окнах, вызываемых, например, с помощью класса `MessageBox`, используются подписи на языке операционной системы (в нашем случае — на русском языке). Кроме того, было бы желательно иметь возможность изменить имя пункта меню, связанного с отображением окна **Select Window**, поскольку в любом неанглоязычном меню пункт **More Windows...** будет выглядеть как инородное включение.

27.4. Одновременное закрытие всех дочерних форм

Добавьте в меню **File** формы `Form2` новый пункт с текстом **`Close &All`**. Данный пункт следует добавить после пункта **Close**; для этого надо выделить разделительную черту, расположенную ниже пункта **Close**, нажать правую кнопку мыши и в появившемся контекстном меню выполнить команду **Insert | MenuItem**. Можно также поступить по-другому: создать пункт **Close All** в конце меню **File**, а затем перетащить его мышью на новое место. Положите свойство `Name` созданного пункта меню равным **`closeAll1`** и определите обработчик события `Click` для этого пункта меню (листинг 27.8).

Листинг 27.8. Обработчик `closeAll1.Click`

```
private void closeAll1_Click(object sender, EventArgs e)
{
  foreach (Form f in MdiParent.MdiChildren)
    f.Close();
}
```

Результат. При выполнении команды **Close All** происходит закрытие всех child-форм.

> ***ПРИМЕЧАНИЕ***
>
> Команда **Close All** включена в меню формы `Form2`, так как она должна быть доступна только при наличии в приложении открытых child-форм.

Комментарий

В методе `closeAll1_Click` использовано свойство-массив `MdiChildren` класса `Form`, содержащее все открытые в данный момент child-формы. Для перебора всех дочерних форм в нашем случае удобно использовать цикл `foreach`. Заметим, что цикл вида

```
for (int i = 0; i < MdiParent.MdiChildren.Length; i++)
  MdiParent.MdiChildren[i].Close();
```

который выполняет, казалось бы, то же самое действие, будет работать неправильно или даже приведет к ошибке времени выполнения. Это связано с тем, что при удалении очередной child-формы массив `MdiChildren` изменяется, а его размер уменьшается, поэтому цикл может завершиться еще до удаления всех child-форм. Однако следующий вариант цикла

```
for (int i = MdiParent.MdiChildren.Length - 1; i >= 0; i++)
  MdiParent.MdiChildren[i].Close();
```

тоже работает неправильно! В этом случае причиной ошибки является то обстоятельство, что после разрушения child-формы, *из которой вызван данный обработчик*, ее свойство `MdiParent` станет недействительным, и попытка обращения к нему приведет к ошибке времени выполнения. Правильно будет работать вариант цикла, в котором используется вспомогательная переменная для хранения значения свойства `MdiParent`:

```
Form f = MdiParent;
for (int i = f.MdiChildren.Length - 1; i >= 0; i++)
  f.MdiChildren[i].Close();
```

Осталось объяснить, почему правильно работает вариант с циклом `foreach`. Перед выполнением этого цикла ссылка на массив `MdiParent.MdiChildren` сохраняется во вспомогательной переменной, к которой и выполняется обращение на каждой итерации цикла. Изменение свойства `MdiChildren` в этой ситуации не может нарушить нормальную работу цикла `foreach`. Действительно, при изменении состава child-форм создается *новый массив* с оставшимися child-формами, ссылка на который записывается в свойство `MdiChildren`, стирая предыдущую ссылку, однако подобные действия не могут затронуть "старый" массив child-форм, ссылка на который хранится во временной переменной, созданной при запуске цикла `foreach`.

27.5. Масштабирование изображения

Добавьте в меню **File** формы `Form2` новый пункт с текстом **`&Stretch`**. Данный пункт надо добавить после пункта **Close All**, используя один из способов, описанных в начале *разд. 27.4*. Положите свойство `Name` созданного пункта меню равным **`stretch1`**, свойство `CheckOnClick` равным **`True`** и определите обработчик события `CheckedChanged` для этого пункта меню (листинг 27.9).

Листинг 27.9. Обработчик `stretch1.CheckedChanged`

```
private void stretch1_CheckedChanged(object sender, EventArgs e)
{
  if (stretch1.Checked)
  {
    pictureBox1.Dock = DockStyle.Fill;
    pictureBox1.SizeMode = PictureBoxSizeMode.Zoom;
  }
  else
  {
    pictureBox1.Dock = DockStyle.None;
    pictureBox1.SizeMode = PictureBoxSizeMode.AutoSize;
  }
}
```

Результат. Изображение можно масштабировать в соответствии с текущими размерами содержащей его child-формы (при этом сохраняются пропорции изображения). Для включения/выключения режима масштабирования предусмотрена команда-переключатель **Stretch**. Если режим масштабирова-

ния включен, то около этой команды в меню изображается "галочка". При отключенном режиме масштабирования устанавливается исходный размер изображения; на child-форме в этом случае могут появиться полосы прокрутки. В каждой child-форме режим масштабирования устанавливается независимо; состояние команды-переключателя **Stretch** соответствует режиму для активной в данный момент child-формы.

Комментарий

Для включения режима масштабирования необходимо "растянуть" компонент `pictureBox1` по child-форме (положив его свойство `Dock` равным `DockStyle.Fill`) и установить для него режим отображения `Zoom` (изображение масштабируется относительно размеров компонента с сохранением пропорций). Для отключения режима масштабирования надо отменить для компонента `pictureBox1` режим стыковки `Fill` и установить для него режим отображения `AutoSize` (размер компонента подстраивается под размер изображения).

27.6. Автоматическая корректировка размера дочерних форм

Добавьте в меню **File** формы `Form2` два новых пункта с текстом **`&Resize`** и **`R&esize All`**. Новые пункты надо добавить после пункта **Stretch**, используя один из способов, описанных в начале *разд. 27.4*. Положите свойства `Name` созданных пунктов меню равными **`resize1`** и **`resizeAll1`** соответственно. Окончательный вид меню группы **File** для формы `Form2` приводится на рис. 27.4.

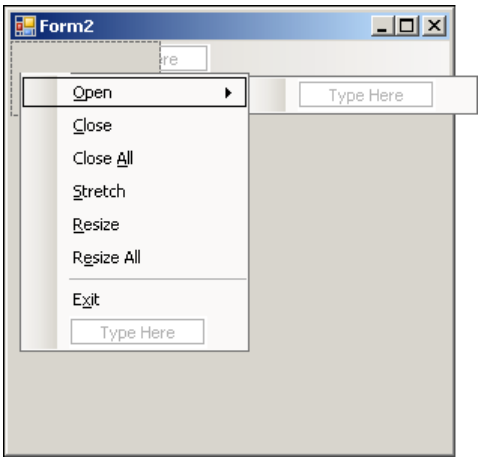


Рис. 27.4. Окончательный вид формы `Form2` для проекта JPEGVIEW

Определите обработчики события `Click` для новых пунктов меню (листинг 27.10).

Листинг 27.10. Обработчики `resize1.Click` и `resizeAll1.Click`

```
private void resize1_Click(object sender, EventArgs e)
{
  Size sz = ClientSize;
  if (stretch1.Checked)
  {
    int iw = pictureBox1.Image.Width,
      ih = pictureBox1.Image.Height,
      cw = ClientSize.Width,
      ch = ClientSize.Height;
    double x = cw / (double)iw,
      y = ch / (double)ih;
    if (x < y)
      sz.Height = (int)(x * ih);
    else
      sz.Width = (int)(y * iw);
  }
  else
    sz = pictureBox1.Size;
  ClientSize = sz;
}
private void resizeAll1_Click(object sender, EventArgs e)
{
  MdiParent.LayoutMdi(MdiLayout.Cascade);
  foreach (Form2 f in MdiParent.MdiChildren)
    f.resize1_Click(f, null);
}
```

Результат. Команда **Resize** настраивает размеры активной child-формы в соответствии с текущими размерами содержащегося в ней изображения. Команда **Resize All** обеспечивает настройку размеров всех child-форм; при этом child-формы размещаются каскадом.

ПРИМЕЧАНИЕ

Вызов метода `LayoutMdi`, обеспечивающего размещение дочерних форм каскадом, следует выполнять *до* настройки размеров, так как метод `LayoutMdi` сам изменяет размеры child-форм.

Комментарии

1. Если режим масштабирования отключен, то настройка размеров формы является тривиальной задачей: достаточно установить размер клиентской части формы равным размеру компонента `pictureBox1`. В режиме масштабирования так поступить нельзя, поскольку размер компонента `pictureBox1` не равен размеру отмасштабированного изображения. Однако можно воспользоваться тем обстоятельством, что в режиме масштабирования `Zoom` достаточно откорректировать размер формы только по *одному* измерению (по ширине или по высоте). Для определения того, какое измерение и на какую величину откорректировать, в методе `resize1_Click` сравниваются пропорции реального (неотмасштабированного) изображения и клиентской части формы.
2. Обратите внимание на то, что в методе `resizeAll1_Click` в качестве типа параметра `f` цикла `foreach` указан класс `Form2`. Это не приведет к ошибке, так как все child-формы нашего приложения действительно имеют тип `Form2`. Если бы для параметра цикла `f` был указан базовый тип `Form`, то при вызове метода `resize1_Click` потребовалось бы использовать операцию приведения типа `as`:

```
(f as Form2).resize1_Click(f, null);
```

27.7. Дополнительные средства управления дочерними формами

Разместите в форме `Form1` *панель инструментов* — компонент типа `ToolStrip` (этот компонент получит имя `toolStrip1`). Панель инструментов автоматически пристыкуется к верхней границе формы и будет располагаться ниже главного меню `menuStrip1`.

Действуя таким же образом, как в *разд. 21.1*, добавьте на панель инструментов `toolStrip1` четыре компонента-кнопки типа `ToolStripButton` и измените их имена (т. е. свойства `Name`) на следующие: **`open2`**, **`close2`**, **`resize2`**, **`stretch2`**, а свойства `Text` положите равными **`Open`**, **`Close`**, **`Resize`** и **`Stretch`** соответственно. Для всех четырех кнопок установите значение свойства `DisplayStyle` равным **`Text`**, значение `AutoSize` равным **`False`** и значение `Size.Width` рав-

ным **50** (последние две настройки необходимы для установки одинаковой ширины кнопок, так как по умолчанию ширина кнопки определяется размером ее надписи). Вид полученной панели инструментов приведен на рис. 27.5.

Измените модификаторы доступа для двух членов класса `Form2`: для пункта меню `stretch1` положите значение свойства `Modifiers` равным **`Internal`**, для метода `resize1_Click` (см. листинг 27.10) замените в его заголовке модификатор `private` на **`internal`**.

Вернитесь к форме `Form1` и свяжите событие `Click` кнопки `open2` с существующим обработчиком `open1_Click`, а также определите обработчики события `Click` для кнопок `close2`, `resize2` и `stretch2` (листинг 27.11).

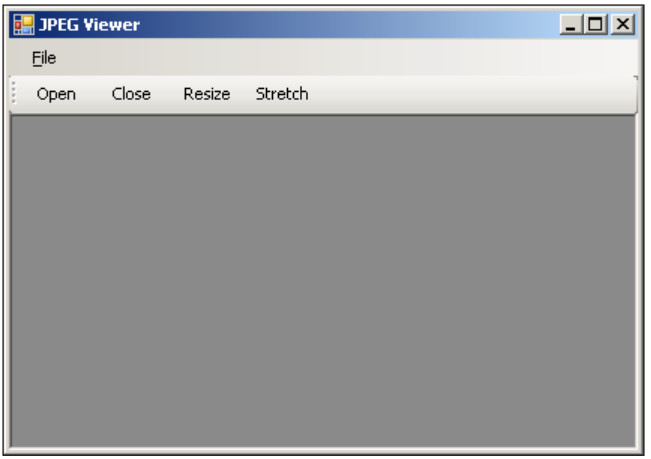


Рис. 27.5. Окончательный вид формы `Form1` для проекта JPEGVIEW

Листинг 27.11. Обработчики `close2.Click`, `resize2.Click` и `stretch2.Click`

```
private void close2_Click(object sender, EventArgs e)
{
  ActiveMdiChild.Close();
}
private void resize2_Click(object sender, EventArgs e)
{
  (ActiveMdiChild as Form2).resize1_Click(this, null);
}
private void stretch2_Click(object sender, EventArgs e)
{
```

```
  ToolStripMenuItem mi = (ActiveMdiChild as Form2).stretch1;
  mi.Checked = !mi.Checked;
}
```

Результат. Кнопка быстрого доступа **Open** дублирует одноименную команду меню. Кнопки **Close** и **Resize** обеспечивают закрытие и настройку размера активной child-формы, а кнопка-переключатель **Stretch** — включение и выключение масштабирования изображения в активной child-форме.

> ***ПРИМЕЧАНИЕ***
>
> При реализации действий по нажатию кнопок **Close**, **Resize** и **Stretch** используется свойство `ActiveMdiChild` главной формы, возвращающее активную child-форму.

Ошибка. Нажатие кнопок **Close**, **Resize** или **Stretch** при *отсутствии* child-форм приводит к исключительной ситуации `NullReferenceException` (поскольку в этом случае свойство `ActiveMDIChild` равно `null`).

Исправление. Для кнопок `close2`, `resize2` и `stretch2` положите свойство `Modifiers` равным **`Internal`**, свойство `Enabled` равным **`False`** и измените метод `open1_Click` формы `Form1` (листинг 27.12) и метод `Form2_FormClosed` формы `Form2` (листинг 27.13).

Листинг 27.12. Новый вариант метода `open1_Click` формы `Form1`

```
internal void open1_Click(object sender, EventArgs e)
{
  openFileDialog1.FileName = "";
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    Form2 f = new Form2();
    string n = openFileDialog1.FileName;
    Image i = Image.FromFile(n);
    f.pictureBox1.Image = i;
    f.Text = string.Format("{0} ({1}x{2})",
      System.IO.Path.GetFileNameWithoutExtension(n), i.Width, i.Height);
    f.MdiParent = this;
    window1.Visible = true;
    close2.Enabled = resize2.Enabled = stretch2.Enabled = true;
    f.Show();
  }
}
```

Листинг 27.13. Новый вариант метода Form2_FormClosed формы Form2

```
private void Form2_FormClosed(object sender, FormClosedEventArgs e)
{
  Form1 f = MdiParent as Form1;
  if (f.MdiChildren.Length == 1)
    f.window1.Visible = f.close2.Enabled = f.resize2.Enabled =
      f.stretch2.Enabled = false;
}
```

Результат. Если главная форма не содержит ни одной child-формы, то кнопки **Close**, **Resize** и **Stretch** недоступны. Отметим, что скрывать недоступные кнопки быстрого доступа (в отличие от пунктов меню первого уровня) не принято.

Недочет 1. При наведении курсора мыши на любую кнопку быстрого доступа рядом с курсором отображается всплывающая подсказка, дублирующая текст на кнопке.

Исправление. В нашем случае нет необходимости в отображении всплывающих подсказок, поскольку сами кнопки содержат текст, поясняющий их назначение. Поэтому отключим возможность отображения всплывающих подсказок для всей панели инструментов, положив свойство `ShowItemToolTips` компонента `toolStrip1` равным **`False`** (см. также комментарий 1).

Недочет 2. Для большей наглядности желательно отображать наличие или отсутствие режима масштабирования с помощью выделения кнопки **Stretch**.

Исправление. Определите обработчик события `Enter` для формы `Form2` (листинг 27.14).

В метод `stretch1_CheckedChanged`, описанный в листинге 27.9, добавьте оператор:

```
(MdiParent as Form1).stretch2.Checked = stretch1.Checked;
```

Последний оператор в методе `Form2_FormClosed`, описанном в листинг 27.13, дополните следующим образом:

```
f.window1.Visible = f.close2.Enabled = f.resize2.Enabled =
  f.stretch2.Enabled = f.stretch2.Checked = false;
```

ПРИМЕЧАНИЕ

Изменение в методе `Form2_FormClosed` гарантирует, что *недоступная* кнопка **Stretch** никогда не будет изображаться выделенной.

Листинг 27.14. Обработчик `Form2.Enter`

```
private void Form2_Enter(object sender, EventArgs e)
{
  (MdiParent as Form1).stretch2.Checked = stretch1.Checked;
}
```

Результат. Теперь переход кнопки **Stretch** из выделенного в невыделенное состояние и обратно происходит в следующих ситуациях (см. также комментарий 2):

- при щелчке мышью на этой кнопке;
- при выполнении команды меню **Stretch**;
- при переходе к дочерней форме, имеющей другой режим масштабирования.

Комментарии

1. Если требуется отключить всплывающую подсказку для какой-либо отдельной кнопки панели `ToolStrip`, то для этой кнопки следует положить свойство `AutoToolTip` равным **`False`** и проверить, что ее свойство `ToolTipText` является пустым. Заметим, что если свойство `AutoToolTip` равно **`True`** (это значение по умолчанию для компонента `ToolStripButton`), то свойство `ToolTipText` всегда имеет значение, совпадающее со свойством `Text`.
2. Обратите внимание на то, что сама кнопка `stretch2` при нажатии на нее не изменяет свое состояние: она только изменяет состояние пункта меню `stretch1` (см. листинг 27.11), однако в результате этого изменения вызывается обработчик `stretch1_CheckedChanged`, который и изменяет состояние кнопки. Это наиболее простой способ взаимодействия кнопки и пункта меню, при котором состояние кнопки должно изменяться и при нажатии на нее, и при выборе связанного с ней пункта меню.

27.8. Прокрутка изображения с помощью клавиатуры

Определите обработчик события `KeyDown` для формы `Form2` (листинг 27.15).

Листинг 27.15. Обработчик `Form2.KeyDown`

```
private void Form2_KeyDown(object sender, KeyEventArgs e)
{
  if (e.Modifiers != Keys.Control)
```

```
    return;
  Point p = AutoScrollPosition;
  Size s = pictureBox1.Size;
  p.X = -p.X;
  p.Y = -p.Y;
  switch (e.KeyCode)
  {
    case Keys.Up:
      p -= new Size(0, 10); break;
    case Keys.Down:
      p += new Size(0, 10); break;
    case Keys.Left:
      p -= new Size(10, 0); break;
    case Keys.Right:
      p += new Size(10, 0); break;
    case Keys.Home:
      p = Point.Empty; break;
    case Keys.PageDown:
      p = new Point(s); break;
    case Keys.End:
      p = new Point(0, s.Height); break;
    case Keys.PageUp:
      p = new Point(s.Width, 0); break;
  }
  AutoScrollPosition = p;
}
```

Результат. Прокрутку изображения (если режим масштабирования отключен и child-форма содержит лишь часть изображения) можно выполнять не только мышью на полосах прокрутки, но и с помощью клавиатуры при нажатой клавише <Ctrl>: клавиши со стрелками обеспечивают прокрутку в указанном направлении, а клавиши <Home>, <End>, <PgUp> и <PgDn> обеспечивают перемещение в один из углов изображения (соответственно, в левый верхний, левый нижний, правый верхний и правый нижний). Необходимость в использовании клавиши <Ctrl> объясняется тем, что обычные клавиши со стрелками уже зарезервированы в MDI-приложении для переключения между дочерними формами *(см. разд. 27.1)*.

Приведем пример работающей программы (рис. 27.6). В окне программы содержатся две дочерние формы с одним и тем же изображением. В левой (активной) форме включен режим масштабирования, правая снабжена полосами прокрутки.

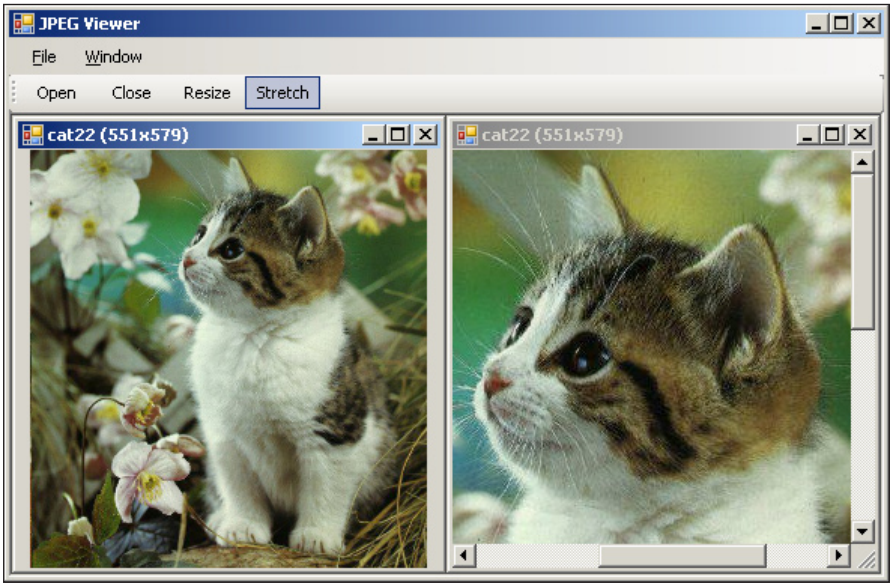


Рис. 27.6. Вид работающего приложения JPEGVIEW

Комментарий

Свойство `AutoScrollPosition`, имеющееся у формы (а также у любого визуального компонента — потомка класса `ScrollableControl`, например, у компонента `Panel`), позволяет программно управлять прокруткой содержимого формы. Следует иметь в виду, что данное свойство, подобно свойству `SelectedText` компонента `TextBox` (см. комментарий 2 в *разд. 8.1*) и отчасти свойству `SelectedIndex` компонента `ListBox` (см. комментарий 1 в *разд. 26.2*), относится к группе свойств с не вполне естественным поведением: если присвоить свойству `AutoScrollPosition` значение некоторого объекта `p` типа `Point` с *положительными* полями `X` и `Y` (задающими позиции полос прокрутки), а затем прочесть новое значение этого свойства, то будет возвращен объект типа `Point` с *отрицательными* полями `X` и `Y` (точнее, значения полей будут *противоположны* значениям `p.X` и `p.Y`). Поэтому, чтобы обеспечить правильную работу нашей программы, в начале метода `Form2_KeyDown` (см. листинг 27.15) приходится использовать операторы `p.X = -p.X` и `p.Y = -p.Y`.

Глава 28

Таблица с текстовыми данными: проект FONTVIEW

Проект FONTVIEW посвящен приемам работы со шрифтами и их семействами (классы `Font` и `FontFamily`, а также компонент `FontDialog`). В нем описываются способы получения списка доступных шрифтов, определения доступных стилей для указанного шрифта, установки размера шрифта с помощью различных единиц измерения. Дается краткое описание кодировки Unicode, и рассматриваются особенности перевода в эту кодировку "традиционных" шрифтов, содержащих 255 символов. Также приводятся начальные сведения по использованию компонента-таблицы `DataGridView`.

28.1. Получение списка доступных шрифтов

После создания проекта FONTVIEW разместите в форме `Form1` компонент типа `SplitContainer` (он получит имя `splitContainer1`), и на левой панели `Panel1` данного компонента разместите компонент-список `listBox1`. Настройте свойства формы и добавленных компонентов (листинг 28.1; рис. 28.1).

Дополните конструктор класса `Form1` (листинг 28.2).

Листинг 28.1. Настройка свойств

```
Form1: Text = Font Viewer, StartPosition = CenterScreen
splitContainer1: FixedPanel = Panel1
listBox1: Dock = Fill, IntegralHeight = False
```

Листинг 28.2. Новый вариант конструктора формы `Form1`

```
public Form1()
{
  InitializeComponent();
  foreach (FontFamily ff in FontFamily.Families)
```

```
    listBox1.Items.Add(ff.Name);
  listBox1.SelectedIndex = listBox1.Items.IndexOf(Font.Name);
}
```

Результат. После запуска программы в списке `listBox1` отображаются названия всех шрифтов, которые можно использовать в графических Windows-приложениях. При этом выделенным является шрифт, заданный в свойстве `Font` формы (по умолчанию это **`Microsoft Sans Serif`**).

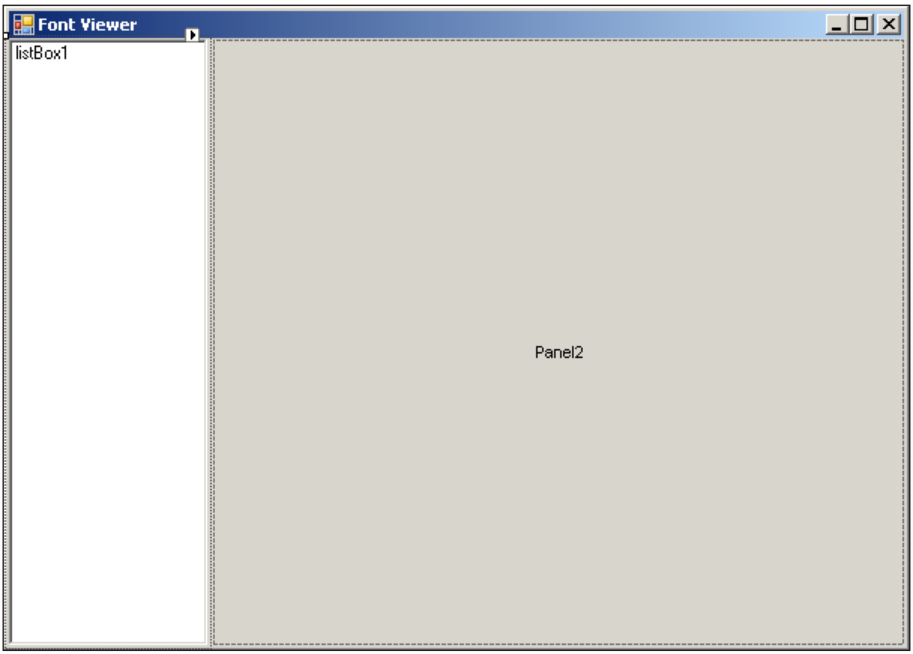


Рис. 28.1. Вид формы `Form1` для проекта FONTVIEW на начальном этапе разработки

Комментарии

1. Набор всех шрифтов можно получить с помощью статического свойства-массива `Families` класса `FontFamily`. Заметим, что данный массив содержит не имена шрифтов, а объекты типа `FontFamily`, поэтому для получения строковых имен приходится дополнительно использовать свойство `Name` этих объектов.
2. Благодаря установке свойства `FixedPanel` компонента `splitContainer1` равным **`Panel1`**, при изменении ширины формы ширина левой панели ком-

понента splitContainer1 (содержащей список listBox1) не изменяется. В то же время, ширину этой панели можно изменить с помощью мыши, перетаскивая разделитель, расположенный между панелями (при перемещении курсора мыши на этот разделитель курсор принимает вид двунаправленной стрелки).

3. Напомним, что установка свойства IntegralHeight компонента listBox1 равным **False** обеспечивает вывод элемента в нижней части списка, даже если этот элемент не может быть целиком отображен на экране (см. комментарий 3 в *разд. 13.2*).

28.2. Отображение символов шрифта в таблице

На правой панели Panel2 компонента splitContainer1 разместите еще один компонент типа SplitContainer (он получит имя splitContainer2). Для компонента splitContainer2 установите свойство Orientation равным **Horizontal**; в результате панель Panel1 компонента splitContainer2 будет размещена в его верхней части, а панель Panel2 — в нижней. Зафиксируйте положение верхней панели, положив свойство FixedPanel компонента splitContainer2 равным **Panel1.**

На верхней панели компонента splitContainer2 разместите два компонента: компонент-панель типа Panel и компонент-таблицу типа DataGridView (новые компоненты получат имена panel1 и dataGridView1). Установите свойство Dock компонента panel1 равным **Top** и разместите в этом компоненте метку label1 (текст метки изменять не требуется). Настройте свойства таблицы dataGridView1 (листинг 28.3; рис. 28.2).

Измените конструктор класса Form1 (листинг 28.4) и определите обработчики события SelectedIndexChanged для компонента listBox1 и события CurrentCellChanged для компонента dataGridView1 (листинг 28.5).

Листинг 28.3. Настройка свойств

```
dataGridView1: Dock = Fill, AllowUserToResizeColumns = False,
  AllowUserToResizeRows = False, AutoSizeColumnsMode = Fill,
  ColumnHeadersVisible = False, RowHeadersVisible = False,
  MultiSelect = False, ReadOnly = True
```

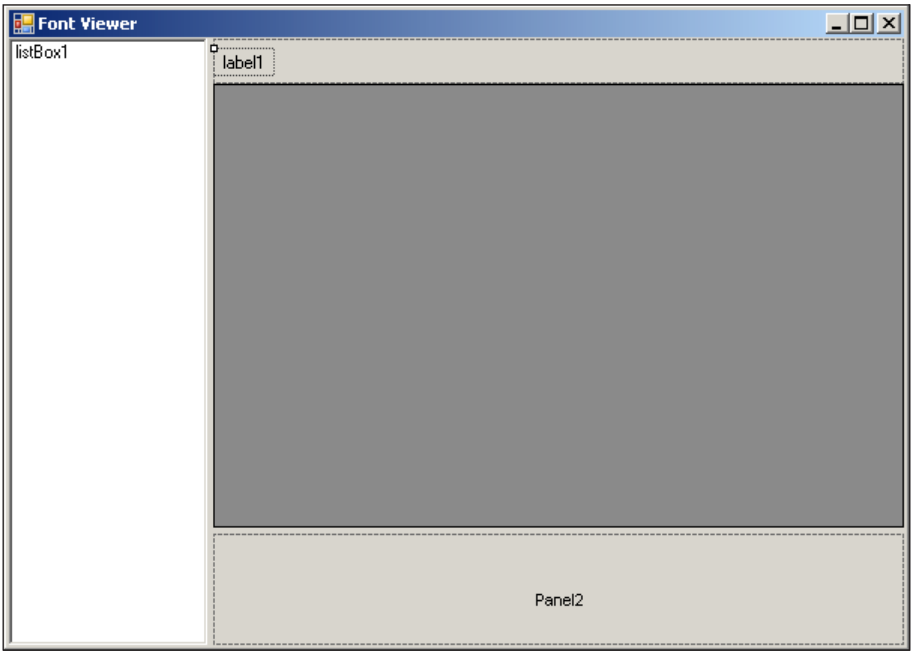


Рис. 28.2. Вид формы `Form1` для проекта FONTVIEW на промежуточном этапе разработки

Листинг 28.4. Новый вариант конструктора формы `Form1`

```
public Form1()
{
  InitializeComponent();
  dataGridView1.ColumnCount = 32;
  dataGridView1.RowCount = 36;
  for (int row = 0; row < 36; row++)
    for (int col = 0; col < 32; col++)
      dataGridView1[col, row].Value =
        ((char)(32 + col + row * 32)).ToString();
  foreach (FontFamily ff in FontFamily.Families)
    listBox1.Items.Add(ff.Name);
  listBox1.SelectedIndex = listBox1.Items.IndexOf(Font.Name);
}
```

Листинг 28.5. Обработчики `listBox1.SelectedIndexChanged` и `dataGridView1.CurrentCellChanged`

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  Font f = dataGridView1.Font;
  dataGridView1.Font = new Font(listBox1.Text, f.Size);
  f.Dispose();
}
private void dataGridView1_CurrentCellChanged(object sender, EventArgs e)
{
  string s = dataGridView1.CurrentCell.Value.ToString();
  label1.Text =
    string.Format("Symbol: {0} Code: {1:X} ({1})", s, (int)s[0]);
}
```

Результат. В таблице, расположенной в правой части формы, отображаются символы текущего (т. е. выделенного в списке `listBox1`) шрифта, начиная с символа пробела, имеющего код 32, и заканчивая символом с кодом 1183. Каждая строка таблицы содержит 32 символа; число строк (36) выбрано таким образом, чтобы в таблицу попали все русские буквы, которые находятся в диапазоне кодов от 1040 до 1103, за исключением букв "Ё" и "ё", имеющих коды 1025 и 1105 соответственно (в указанном диапазоне вначале располагаются в алфавитном порядке прописные буквы, а затем — строчные).

Текущий символ таблицы вместе с его кодовым номером в шестнадцатеричном и десятичном представлении (например, **Symbol: Z Code: 5A (90)**) указывается в метке `label1` на панели, расположенной над таблицей символов. Для текста метки `label1` используется стандартный шрифт Microsoft Sans Serif.

При изменении ширины формы ширина всех столбцов таблицы изменяется таким образом, чтобы все столбцы отображалась на экране и заполняли всю область таблицы.

Ошибка 1. При выделении в таблице символа **&** (десятичный кодовый номер 38) в метке `label1` вместо данного символа отображается символ подчеркивания. Возможно также, что в указанной ситуации будет изображен обычный пробел; в этом случае для вывода символа подчеркивания достаточно нажать клавишу <Alt>.

Исправление. Положите свойство `UseMnemonic` метки `label1` равным **`False`**.

ПРИМЕЧАНИЕ

Свойство `UseMnemonic` имеется не только у компонентов-меток `Label`, но и у кнопок `Button`, флажков `CheckBox` и радиокнопок `RadioButton`. Если это свойство равно **`True`** (значение по умолчанию), то символ `&` в свойстве `Text` компонента интерпретируется как *управляющий символ*, который означает, что символ, следующий за ним, должен быть подчеркнут.

Ошибка 2. При выделении в списке `listBox1` некоторых шрифтов (например, Monotype Corsiva) в программе возникает ошибка времени выполнения с сообщением "Font <*имя шрифта*> does not support style 'Regular'". Эта ошибка связана с тем обстоятельством, что некоторые шрифты не имеют реализации для обычного стиля (Regular), а именно этот стиль устанавливается по умолчанию при создании нового шрифта в обработчике `listBox1_SelectedIndexChanged` (см. листинг 28.5). Мы исправим эту ошибку в следующем разделе.

Комментарии

1. Программа позволяет ознакомиться о особенностями кодировки Unicode, используемой в .NET Framework. В данной кодировке для хранения каждого символа отводится по два байта, что позволяет хранить в одной кодовой таблице символы всех распространенных алфавитов. При этом символы с кодами от 1 до 127 совпадают со стандартной ASCII-кодировкой, а символы с кодами от 128 до 2047 используются для хранения европейских и среднеазиатских языков. Оставшаяся часть кодовой таблицы Unicode используется, в основном, для хранения символов из восточноазиатских языков, в том числе иероглифов, причем применяемый в ней специальный механизм так называемых *символов-заместителей* (surrogate characters) позволяет закодировать более миллиона символов.

   Просмотр различных шрифтов показывает, что многие шрифты реализуют только часть кодировки Unicode. В этом случае часть таблицы будет занята "пустыми" элементами, не имеющими символьного представления (на месте этих элементов изображается белый прямоугольник □). В некоторых специальных шрифтах реализовано всего 255 символов; в этом случае символы шрифта "переносятся" на таблицу Unicode особым образом, занимая первые 127 позиций, а также определенные диапазоны в последующей части таблицы Unicode (в частности, заполняется диапазон, соответствующий русским буквам). Таковы, например, шрифты Symbol, Wingdings, Webdings.

2. Для отображения символов в виде таблицы мы воспользовались компонентом `DataGridView`, появившимся в .NET Framework 2.0. Этот компонент

является одним из наиболее сложных компонентов стандартной библиотеки классов .NET. Достаточно отметить, что он реализует 118 собственных событий (в дополнение к 69 событиям, унаследованным от класса `Control`) и что для работы с его строками, столбцами и ячейками предусмотрена целая иерархия вспомогательных классов, в частности, шесть классов для ячеек с различными типами данных. Имена всех этих вспомогательных классов начинаются с префикса `DataGridView`, например, `DataGridViewColumn` — класс, отвечающий за настройку свойств одного из столбцов таблицы, или `DataGridViewTextBoxCell` — класс, отвечающий за настройку свойств ячейки с текстовой информацией (по умолчанию все ячейки таблицы являются объектами класса `DataGridViewTextBoxCell`).

Несмотря на сложность компонента `DataGridView`, для создания простых таблиц достаточно установить или изменить всего несколько его свойств, среди которых основными являются `ColumnCount` (число столбцов), `RowCount` (число строк) и двумерный *индексатор* с параметрами `[col, row]`, позволяющий задать значение ячейки, расположенной в столбце `col` и строке `row` (все эти свойства доступны и для чтения, и для записи). Заметим, что данный индексатор возвращает значение типа `DataGridViewCell` — общего предка всех классов, связанных с ячейками таблицы. Доступ к значению, содержащемуся в ячейке, обеспечивает свойство `Value` класса `DataGridViewCell`.

3. При настройке свойств нашей простейшей таблицы (см. листинг 28.3) мы скрыли заголовки строк и столбцов (свойства `ColumnHeadersVisible` и `RowHeadersVisible`), запретили явную настройку ширины столбцов и высоты строк (свойства `AllowUserToResizeColumns` и `AllowUserToResizeRows`), а также запретили одновременное выделение нескольких ячеек (свойство `MultiSelect`) и редактирование содержимого таблицы (свойство `ReadOnly`). Кроме того, установив свойство `AutoSizeColumnsMode` равным `Fill`, мы обеспечили "растяжение" таблицы по ширине компонента `DataGridView` (при подобном растяжении таблицы ширина всех ее столбцов изменяется одинаковым образом). Дополнительные возможности, связанные с компонентом `DataGridView`, будут описаны в *главах 29—32*.

28.3. Настройка стиля для выбранного шрифта

В компоненте `panel1` справа от имеющейся метки `label1` разместите три радиокнопки `radioButton1`—`radioButton3` и настройте их свойства (листинг 28.6; рис. 28.3).

Измените метод `listBox1_SelectedIndexChanged` (листинг 28.7).

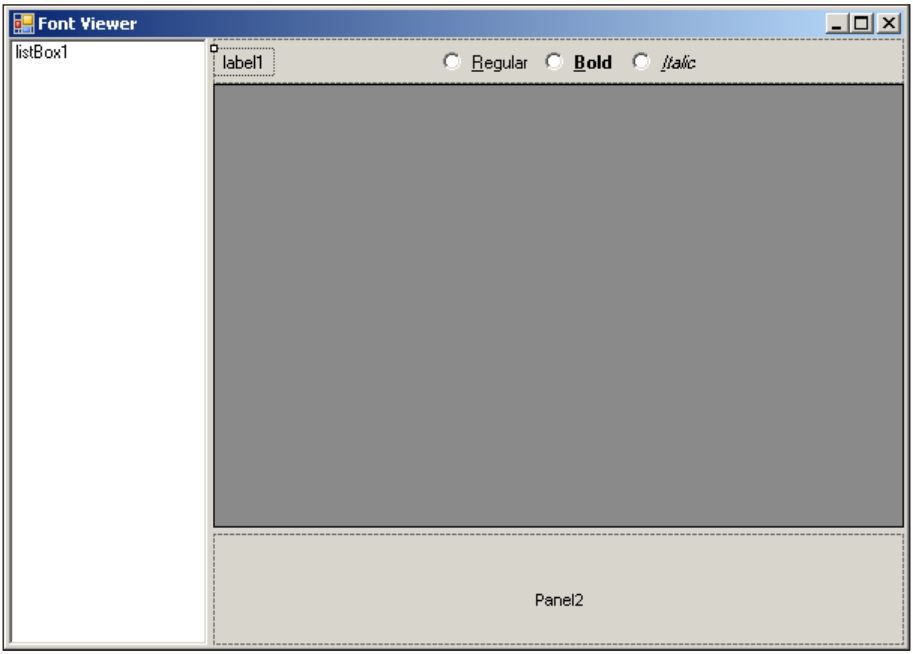


Рис. 28.3. Вид формы `Form1` для проекта FONTVIEW после добавления набора радиокнопок

Определите обработчик события `CheckedChanged` для радиокнопки `radioButton1` (листинг 28.8), после чего свяжите созданный обработчик с событием `CheckedChanged` радиокнопок `radioButton2` и `radioButton3`.

Листинг 28.6. Настройка свойств

```
radioButton1: Text = &Regular
radioButton2: Text = &Bold, Font.Bold = True
radioButton3: Text = &Italic, Font.Italic = True
```

Листинг 28.7. Новый вариант метода `listBox1_SelectedIndexChanged`

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  FontFamily ff = new FontFamily(listBox1.Text);
  bool isChecked = false;
  for (int i = 1; i <= 3; i++)
  {
```

```
    RadioButton rb = panel1.Controls["radioButton" + i] as RadioButton;
    rb.Checked = false;
    rb.Enabled = ff.IsStyleAvailable(rb.Font.Style);
    if (rb.Enabled && !isChecked)
      isChecked = rb.Checked = true;
  }
  ff.Dispose();
}
```

Листинг 28.8. Обработчик `radioButton1.CheckedChanged`

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
  RadioButton rb = sender as RadioButton;
  if (!rb.Checked)
    return;
  Font f = dataGridView1.Font;
  dataGridView1.Font =
    new Font(listBox1.Text, rb.Font.Size, rb.Font.Style);
  f.Dispose();
}
```

Результат. При выборе шрифта из списка `listBox1` остаются доступными только те радиокнопки, которые соответствуют доступным стилям для выбранного шрифта, причем первая из доступных радиокнопок оказывается выбранной (например, для шрифта Monotype Corsiva будет доступна единственная радиокнопка, соответствующая стилю Italic, и именно эта радиокнопка будет выбрана). Стиль шрифта в таблице `dataGridView1` зависит от того, какая из радиокнопок является выбранной. Для выбора стиля шрифта Regular, Bold, Italic можно использовать клавиатурные комбинации <Alt>+<R>, <Alt>+<B>, <Alt>+<I> соответственно.

Комментарии

1. Для определения стилей, доступных для указанного шрифта (точнее, для *семейства шрифтов* — объекта типа `FontFamily`), в обработчике `listBox1_SelectedIndexChanged` (см. листинг 28.7) был использован метод `IsStyleAvailable` класса `FontFamily`. Обратите внимание на то, что для

созданного в обработчике объекта `ff` типа `FontFamily` необходимо вызвать метод `Dispose`, который освободит ресурсы, выделенные системой для этого объекта.

2. В новом варианте программы настройка шрифта для таблицы `dataGridView1` выполняется в обработчике `radioButton1_CheckedChanged` (см. листинг 28.8); при этом в конструкторе класса `Font` явно указывается имя шрифта, его размер и его стиль (размер и стиль шрифта устанавливаются равными размеру и стилю шрифта соответствующей радиокнопки).

28.4. Просмотр текущего символа в увеличенном виде

На нижней панели компонента `splitContainer2` (эта панель имеет имя `Panel2` — см. рис. 28.3) разместите метку `label2` и настройте ее свойства (листинг 28.9). Определите обработчик события `Resize` для панели `Panel2` компонента `splitContainer2` (листинг 28.10). Напомним, что для выделения этой панели достаточно выделить находящуюся на ней метку `label2`, после чего нажать клавишу <Esc>.

В метод `dataGridView1_CurrentCellChanged` добавьте оператор:

```
label2.Text = s;
```

В метод `radioButton1_CheckedChanged` добавьте оператор:

```
splitContainer2_Panel2_Resize(this, null);
```

Листинг 28.9. Настройка свойств

```
label2: Text = пустая строка, AutoSize = False, Dock = Fill,
  TextAlign = MiddleCenter, UseMnemonic = False
```

Листинг 28.10. Обработчик `splitContainer2.Panel2.Resize`

```
private void splitContainer2_Panel2_Resize(object sender, EventArgs e)
{
  label2.Font = new Font(listBox1.Text,
    splitContainer2.Panel2.Height * 3 / 4, dataGridView1.Font.Style,
    GraphicsUnit.Pixel);
}
```

Результат. Текущий символ таблицы отображается в увеличенном виде на нижней панели компонента `splitContainer2`, причем размер символа определяется текущей высотой панели. При выборе другого шрифта или изменении его стиля вид увеличенного символа корректируется. См. также комментарии 1 и 2.

Недочет. При выполнении метода `splitContainer2_Panel2_Resize` не вызывается метод `Dispose` для прежнего шрифта метки `label2`.

Исправление. В начало метода `splitContainer2_Panel2_Resize` вставьте оператор

```
label2.Font.Dispose();
```

Ошибка. При запуске измененного варианта программы немедленно возникает исключение типа `ArgumentException`, причем в качестве оператора, вызвавшего это исключение, указывается следующий оператор из файла Form1.Designer.cs:

```
this.splitContainer1.Panel1.ResumeLayout(false);
```

Можно предположить, что в методе `ResumeLayout` происходит вызов метода `splitContainer2_Panel2_Resize`, и это каким-то образом приводит к возникновению ошибки. Метод `splitContainer2_Panel2_Resize` может быть вызван только как реакция на событие `Resize` (заметим, что оператор, связывающий данный метод с событием `Resize` компонента `splitContainer2.Panel2`, расположен в файле Form1.Designer.cs перед тем оператором, который вызвал исключение). Поэтому исправим обнаруженную ошибку, "откладывая" определение обработчика события `Resize` до того момента, когда все инициализирующие действия, определенные в файле Form1.Designer.cs, уже будут выполнены.

Исправление. В окне **Properties** для компонента `splitContainer2.Panel2` очистите поле ввода, связанное с событием `Resize`; в конструктор класса `Form1` после оператора

```
InitializeComponent();
```

вставьте оператор, обеспечивающий связывание события `Resize` с обработчиком `splitContainer2_Panel2_Resize`:

```
splitContainer2.Panel2.Resize += splitContainer2_Panel2_Resize;
```

Результат. Теперь при запуске программы ошибки не возникает (см. также комментарий 3).

Приведем изображение работающей программы при выбранном шрифте Wingdings (рис. 28.4).

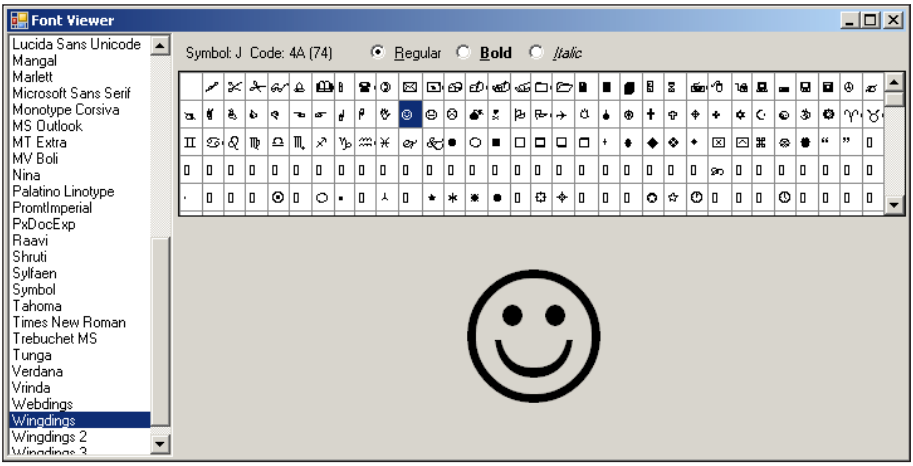


Рис. 28.4. Вид работающего приложения FONTVIEW

Комментарии

1. Для того чтобы размер шрифта соответствовал текущей высоте панели, следует указать этот размер не в пунктах, а в пикселах (т. е. в тех же единицах измерения, в которых задается высота панели). Подобная возможность обеспечивается одним из вариантов конструктора класса `Font`, в котором последний (четвертый) параметр определяет единицу измерения размера шрифта (см. листинг 28.10). Данный параметр имеет перечислимый тип `GraphicsUnit` и может принимать значение `Point` (значение по умолчанию для класса `Font`), `Pixel`, `Inch`, `Millimeter` и некоторые другие. Следует заметить, что различные единицы измерения можно использовать и в методах рисования. Для этого достаточно присвоить свойству `PageUnit` объекта `Graphics` одну из единиц измерения, определенных в перечислении `GraphicsUnit`, а свойству `PageScale` того же объекта `Graphics` — масштабный множитель. Например, после присваиваний `g.PageUnit = GraphicsUnit.Millimeter; g.PageScale = 0.1;` единицей системы координат для объекта `g` типа `Graphics` будет считаться не пиксел, а десятая доля миллиметра. Подробное обсуждение вопросов, связанных с настройкой единиц измерения при рисовании и при работе со шрифтами, содержится в главах 7 и 9 книги [5].

2. Множитель `3/4` в конструкторе класса `Font` (см. листинг 28.10) добавлен для того, чтобы можно было отобразить те фрагменты символа, которые находятся ниже осевой линии текста (так называемые *нижние выносные элементы*, как, например, нижняя часть символа "у").

3. В *разд. 19.4* мы уже сталкивались с аналогичной ошибкой, возникающей при попытке освободить ресурсы для некоторого шрифта. Создается впечатление, что при разработке многих методов стандартной библиотеки, работающих со шрифтами, не учитывалась возможность разрушения "старого" шрифта методом `Dispose` при изменении свойства `Font` какого-либо компонента. Хотя и в *разд. 19.4*, и в настоящем разделе нам удалось исправить ошибку и при этом сохранить фрагменты программы, вызывающие метод `Dispose` для шрифтов, ставших ненужными, следует признать, что исправления были весьма неочевидными. Впрочем, у разработчика всегда остается "последнее средство" для исправления подобных ошибок: отказ от явного разрушения ненужных шрифтов методом `Dispose`.

Глава 29

Таблица с графическими данными: проект ICONVIEW

Проект ICONVIEW посвящен приемам работы со значками (класс `Icon`), связанным с их загрузкой из файлов, отображением на экране и сохранением в различных форматах. В проекте реализуется вспомогательный класс `FileIcons`, позволяющий извлекать значки из exe-, dll- и ico-файлов. При разработке класса `FileIcons` демонстрируется возможность импортирования стандартных функций Win32 API в .NET-приложение (механизм P/Invoke) и способы создания объектной оболочки для импортированной функции. Рассматриваются дополнительные возможности компонента-таблицы `DataGridView`, связанные с отображением графических данных и изменением размеров таблицы в ходе выполнения программы.

29.1. Извлечение значков из файлов

Мы собираемся разработать приложение, которое позволит просматривать значки (пиктограммы), содержащиеся в файлах с расширениями ico, exe и dll (напомним, что ico-файлы специально предназначены для хранения значков, в то время как exe- и dll-файлы могут содержать набор значков в качестве графических ресурсов). Поэтому нам необходим метод, позволяющий извлекать из файла все обнаруженные в нем значки. К сожалению, подобный метод в стандартной библиотеке .NET Framework отсутствует.

> ***ПРИМЕЧАНИЕ***
>
> В версии .NET Framework 2.0 в класс `Icon` добавлен метод `ExtractAssociatedIcon(fileName)`, возвращающий *первый* из значков, содержащихся в файле с именем `fileName`, однако для наших целей возможностей метода `ExtractAssociatedIcon` недостаточно.

В то же время, в системной библиотеке Windows (Win32 API) имеются две функции, предназначенные для извлечения значков из файлов: это `ExtractIcon` и `ExtractIconEx`. Обе функции содержатся в динамической библиотеке Shell32.dll.

Более простой в использовании является функция `ExtractIcon(hInst, fileName, index)`. Параметр `hInst` целого типа задает *дескриптор приложения*, вызвавшего данную функцию (вместо этого параметра допустимо указывать нулевой дескриптор). Параметр `fileName` указывает имя файла, из которого требуется прочесть значок, а `index` — порядковый номер значка в файле (нумерация начинается от 0). Возвращает функция *дескриптор* прочитанного значка (позволяющий в дальнейшем получить доступ к этому значку) или, если параметр `index` равен `-1`, количество значков, обнаруженных в указанном файле.

В .NET Framework предусмотрен механизм Platform Invoke (P/Invoke), позволяющий импортировать любые функции Win32 API (а точнее, любые функции, содержащиеся в динамических библиотеках Windows), в .NET-приложение. Воспользуемся механизмом P/Invoke и импортируем в наш проект функцию `ExtractIcon`. Кроме того, дополнительно определим класс-оболочку для импортированной функции, который упростит ее последующее использование.

После создания проекта ICONVIEW выполните команду меню **Project | Add Class...**; в появившемся диалоговом окне замените имя, предлагаемое по умолчанию, на FileIcons.cs (расширение cs можно не указывать), после чего нажмите клавишу <Enter> или кнопку **Add**. В результате к проекту будет добавлен файл FileIcons.cs с текстом заготовки для класса `FileIcons` (листинг 29.1).

Дополните в файле FileIcons.cs список директив `using` и добавьте в класс `FileIcons` описание необходимых методов (листинг 29.2).

Листинг 29.1. Исходный текст файла FileIcons.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ICONVIEW
{
  class FileIcons
  {
  }
}
```

Листинг 29.2. Новый вариант файла FileIcons.cs

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
using System.Drawing;
using System.Runtime.InteropServices;

namespace ICONVIEW
{
  class FileIcons
  {
    [DllImport("Shell32.dll")]
    private static extern IntPtr ExtractIcon(IntPtr hInst,
      string fileName, int index);
    public static int Count(string fileName)
    {
      return (int)ExtractIcon((IntPtr)0, fileName, -1);
    }
    public static Icon Extract(string fileName, int index)
    {
      try
      {
        return Icon.FromHandle(ExtractIcon((IntPtr)0, fileName, index));
      }
      catch
      {
        return null;
      }
    }
    public static Icon[] ExtractAll(string fileName)
    {
      Icon[] icon = new Icon[FileIcons.Count(fileName)];
      for (int i = 0; i < icon.Length; i++)
        icon[i] = FileIcons.Extract(fileName, i);
      return icon;
    }
  }
}
```

Результат. Созданный класс `FileIcons` содержит три открытых статических метода:

- `Count(`*`fileName`*`)` — позволяет определить количество значков, содержащихся в файле с именем *`fileName`*;
- `Extract(`*`fileName`*`, index)` — возвращает объект типа `Icon`, который содержит значок с индексом *`index`* их файла с именем *`fileName`* (или пустой объект `null`, если значок с требуемым индексом в файле отсутствует или указано недопустимое имя файла);
- `ExtractAll(`*`fileName`*`)` — возвращает массив (типа `Icon[]`) всех значков, найденных в файле с именем *`fileName`* (если файл не найден или не содержит значков, то возвращается массив нулевого размера, т. е. свойство `Length` данного массива будет равно `0`).

Все эти методы основаны на закрытом методе `ExtractIcon`, импортированном из библиотеки Shell32.dll.

Комментарии

1. Для импортирования функции из динамической библиотеки достаточно указать заголовок функции, предварив его атрибутом `DllImport` (в квадратных скобках). Этот атрибут должен содержать один обязательный параметр — имя динамической библиотеки (dll-файла), из которой импортируется указанная функция. Следует обратить внимание на модификатор `extern` в заголовке функции, а также на то, что для дескрипторов в .NET предусмотрен особый тип `IntPtr`. Входные строковые параметры достаточно описывать как `string`. Заметим, что если строковый параметр импортируемой функции является *выходным* (т. е. может изменяться в ходе работы функции), то для него надо использовать тип `StringBuilder`.
2. Для передачи в функцию `ExtractIcon` нулевого дескриптора необходимо привести число `0` к типу `IntPtr`. В методе `Count` требуется применить обратное преобразование (от типа `IntPtr` к типу `int`) для возвращаемого значения функции `ExtractIcon`.
3. Для создания объекта класса `Icon` по дескриптору значка, возвращенному функцией `ExtractIcon`, следует использовать статический метод `FromHandle` класса `Icon`. При недопустимом дескрипторе (который возвращается функцией `ExtractIcon` в случае недопустимого имени файла или неверного индекса значка) метод `FromHandle` генерирует исключение, поэтому метод `Extract` содержит блок `try-catch`, позволяющий перехватить это исключение и возвратить в подобной ситуации значение `null`.

4. Обратите внимание на то, что метод `ExtractAll` никогда не возвращает значение `null`: если файл отсутствует или не содержит значков, то метод вернет массив типа `Icon[]` с нулевым числом элементов. Таким образом, для возвращаемого массива всегда будет доступно свойство `Length` (напомним, что если массив равен `null`, то попытка доступа к его свойству `Length` приведет к возбуждению исключения).

29.2. Загрузка значков и их отображение в таблице

Добавьте в форму `Form1` проекта ICONVIEW панель инструментов `toolStrip1` и таблицу `dataGridView1`, а также невизуальный компонент типа `OpenFileDialog` (этот компонент получит имя `openFileDialog1` и будет размещен в области невизуальных компонентов под изображением формы). Панель инструментов `toolStrip1` будет автоматически пристыкована к верхней границе формы.

Действуя таким же образом, как в *разд. 21.1*, добавьте на панель инструментов `toolStrip1` компонент-кнопку типа `ToolStripButton` и измените его имя (т. е. свойство `Name`) на **`load1`**. Настройте свойства формы `Form1` и добавленных компонентов (листинг 29.3; рис. 29.1).

В описание класса `Form1` добавьте поле

```
private int n = -1;
```

Дополните конструктор класса `Form1` (листинг 29.4) и определите обработчик события `Click` для кнопки `load1` (листинг 29.5).

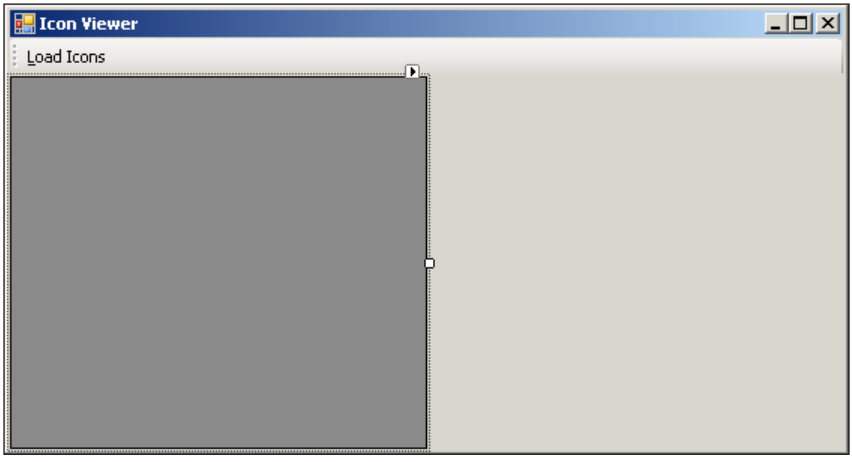


Рис. 29.1. Вид формы `Form1` проекта ICONVIEW на начальном этапе разработки

Листинг 29.3. Настройка свойств

```
Form1: Text = Icon Viewer, StartPosition = CenterScreen
toolStrip1: ShowItemToolTips = False
load1: Text = &Load Icons, DisplayStyle = Text
dataGridView1: Dock = Left, AllowUserToAddRows = False,
  AllowUserToResizeColumns = False, AllowUserToResizeRows = False,
  ColumnHeadersVisible = False, RowHeadersVisible = False,
  MultiSelect = False, ScrollBars = Vertical
openFileDialog1: DefaultExt = ico,  Title = Load icons from file,
  Filter = Files with icons|*.ico;*.exe;*.dll
```

Листинг 29.4. Новый вариант конструктора формы `Form1`

```
public Form1()
{
  InitializeComponent();
  for (int i = 0; i < 10; i++)
  {
    DataGridViewImageColumn c = new DataGridViewImageColumn(true);
    c.Width = 40;
    dataGridView1.Columns.Add(c);
  }
  dataGridView1.Width = 420;
  dataGridView1.RowCount++;
  dataGridView1.Rows[0].Height = 40;
}
```

Листинг 29.5. Обработчик `load1.Click`

```
private void load1_Click(object sender, EventArgs e)
{
  openFileDialog1.FileName = "";
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    Icon[] ic = FileIcons.ExtractAll(openFileDialog1.FileName);
    for (int i = 0; i < ic.Length; i++)
```

```
    {
      n++;
      if (n / 10 == dataGridView1.RowCount)
      {
        dataGridView1.RowCount++;
        dataGridView1.Rows[dataGridView1.RowCount - 1].Height = 40;
      }
      DataGridViewCell c = dataGridView1[n % 10, n / 10];
      c.Value = ic[i];
      if (i == 0)
        c.Selected = true;
    }
  }
}
```

Результат. При запуске программы таблица `dataGridView1` содержит единственную строку из 10 пустых графических ячеек (в пустых ячейках выводится особое изображение). При нажатии кнопки **Load Icons** или клавиатурной комбинации <Alt>+<L> открывается диалоговое окно **Load icons from file**, позволяющее выбрать файл, содержащий значки, т. е. файл с расширением exe, dll или ico (при последующих вызовах этого диалогового окна в нем отображается последний выбранный каталог). Все значки из выбранного файла выводятся в ячейках таблицы `dataGridView1`. При исчерпании ячеек в текущей строке количество строк в таблице увеличивается. Значки из новых файлов добавляются к уже имеющимся в таблице. После добавления значков из выбранного файла автоматически выделяется первый из добавленных значков.

> ***ПРИМЕЧАНИЕ***
>
> Для доступа к значкам, хранящимся в файлах, был использован метод `ExtractAll` класса `FileIcons`, разработанного в *разд. 29.1* (см. листинг 29.2).

Комментарии

1. Ячейки таблицы `DataGridView` могут содержать не только текст, но и данные других типов. Для указания требуемого типа ячейки проще всего задать соответствующий тип для всего столбца таблицы. Например, если столбец имеет тип `DataGridViewTextBoxColumn` (значение по умолчанию), то все ячейки этого столбца будут иметь тип

DataGridViewTextBoxCell, т. е. будут содержать текстовые данные. Можно также создавать столбцы с флажками (DataGridViewCheckBoxColumn), выпадающими списками (DataGridViewComboBoxColumn), изображениями (DataGridViewImageColumn), кнопками (DataGridViewButtonColumn) и гипертекстовыми ссылками (DataGridViewLinkColumn). В нашем случае в конструкторе формы создаются 10 столбцов, содержащих ячейки с изображениями (т. е. ячейки типа DataGridViewImageCell), которые последовательно добавляются к таблице dataGridView1 с помощью команды Add ее свойства-коллекции Columns (см. листинг 29.4). При создании столбцов используется конструктор класса DataGridViewImageColumn с одним необязательным параметром valuesAreIcons логического типа. Этот параметр определяет, в виде какого класса будет храниться содержимое ячеек: если параметр равен true, то в ячейках будут содержаться данные типа Icon (как в нашем случае); если параметр конструктора не указан или равен false, то в ячейках будут содержаться данные типа Image.

2. Одновременно с созданием каждого столбца задается его ширина, равная 40 пикселам (см. листинг 29.4); этого размера достаточно для вывода значка, так как все значки имеют стандартный размер 32 × 32 пиксела. Ширина всего компонента dataGridView1 полагается равной 420 пикселам; это позволяет отобразить на экране не только все 10 столбцов, но и вертикальную полосу прокрутки, которая возникает в случае, если все строки таблицы нельзя одновременно отобразить на экране. Отметим, что вместо явного задания ширины каждого столбца можно было установить свойство AutoSizeColumnsMode компонента dataGridView1 равным **Fill**; в этом случае столбцы заполняли бы по ширине весь компонент dataGridView1, причем при появлении полосы прокрутки ширина столбцов корректировалась бы автоматически.

3. Первая строка добавляется к таблице одновременно с созданием столбцов в конструкторе формы Form1 (см. листинг 29.4; для добавления строки достаточно увеличить значение свойства RowCount на 1). Вначале все ячейки этой строки являются пустыми, т. е. их свойство Value равно null. Новые строки добавляются в ситуации, когда для очередного значка оказывается недостаточно имеющихся ячеек (см. условие n / 10 == dataGridView1.RowCount в листинге 29.5). Высота каждой добавляемой строки полагается равной 40 пикселам; для доступа к строкам используется свойство-коллекция Rows компонента dataGridView1. Отметим, что имеется простая возможность обеспечить "подстройку" высоты каждой строки под размеры содержащихся в ней данных; для этого достаточно установить свойство AutoSizeRowsMode компонента dataGridView1 равным **AllCells**.

4. Используя двумерное свойство-индексатор `[col, row]` компонента `dataGridView1`, можно получить доступ к свойствам существующей ячейки (первый индекс определяет номер столбца, второй индекс — номер строки). Помимо свойства `Value`, определяющего значение ячейки, в методе `load1_Click` (см. листинг 29.5) используется свойство `Selected`, позволяющее сделать указанную ячейку выбранной (т. е. выделенной). Заметим, что при значении свойства `MultiSelect` таблицы, равном **`False`** (как в нашем случае), выделение новой ячейки автоматически снимает выделение с той ячейки, которая была выделена ранее.
5. Следует обратить внимание на свойство `AllowUserToAddRows` таблицы, которое необходимо положить равным **`False`** (см. листинг 29.3). Если для этого свойства оставить значение **`True`**, установленное по умолчанию, то последняя строка таблицы будет считаться особой строкой *NewRow* — строкой для добавления новых данных (для строки NewRow, в отличие от всех других строк, свойство `IsNewRow` возвращает значение `true`). Строка NewRow *автоматически* добавляется к таблице при добавлении к этой таблице первого столбца. Если в дальнейшем в таблицу будет добавлена новая строка (например, в результате увеличения значения свойства `RowCount`), то эта строка будет размещена *перед* строкой NewRow.

29.3. Очистка таблицы

Добавьте на панель инструментов `toolStrip1` новый компонент-кнопку и измените его имя (т. е. свойство `Name`) на **`clear1`**. Настройте свойства кнопок `load1` и `clear1` (листинг 29.6; указанные значения свойств `AutoSize` и `Size.Width` позволяют установить одинаковую ширину для обеих кнопок).

Определите обработчик события `Click` для кнопки `clear1` (листинг 29.7).

Листинг 29.6. Настройка свойств

```
load1: AutoSize = False, Size.Width = 100
clear1: Text = &Clear Grid, DisplayStyle = Text, AutoSize = False,
  Size.Width = 100
```

Листинг 29.7. Обработчик `clear1.Click`

```
private void clear1_Click(object sender, EventArgs e)
{
  for (int col = 0; col < dataGridView1.ColumnCount; col++)
```

```
    for (int row = 0; row < dataGridView1.RowCount; row++)
    {
      Icon ic = dataGridView1[col, row].Value as Icon;
      if (ic != null)
      {
        ic.Dispose();
        dataGridView1[col, row].Value = null;
      }
    }
  dataGridView1.RowCount = 1;
  dataGridView1[0,0].Selected = true;
  n = -1;
}
```

Результат. При нажатии кнопки **Clear Grid** или комбинации клавиш <Alt>+<C> таблица со значками очищается, число ее строк уменьшается до 1, а в оставшейся строке выделяется первая ячейка.

> ***ПРИМЕЧАНИЕ***
>
> Обратите внимание на то, что перед очисткой ячейки (путем присваивания ее свойству `Value` значения `null`) следует освободить ресурсы, выделенные для значка, который изображен в этой ячейке (вызвав для этого значка метод `Dispose`).

29.4. Отображение дополнительной информации о выбранном значке

Разместите в форме `Form1` компонент типа `PictureBox` (он получит имя `pictureBox1`) и настройте его свойства, а также свойство `MinimumSize` формы (листинг 29.8; рис. 29.2).

В начало файла Form1.cs добавьте строку

```
using System.IO;
```

Измените метод `load1_Click` (листинг 29.9) и определите обработчик события `SelectionChanged` для таблицы `dataGridView1` (листинг 29.10).

Листинг 29.8. Настройка свойств

```
Form1: MinimumSize = 480; 100
pictureBox1: Dock = Fill, BorderStyle = FixedSingle, SizeMode = Zoom
```

Рис. 29.2. Вид формы `Form1` проекта ICONVIEW на промежуточном этапе разработки

Листинг 29.9. Новый вариант метода `load1_Click`

```
private void load1_Click(object sender, EventArgs e)
{
  openFileDialog1.FileName = "";
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    Icon[] ic = FileIcons.ExtractAll(openFileDialog1.FileName);
    string name =
      Path.GetFileNameWithoutExtension(openFileDialog1.FileName);
    for (int i = 0; i < ic.Length; i++)
    {
      n++;
      if (n / 10 == dataGridView1.RowCount)
      {
        dataGridView1.RowCount++;
        dataGridView1.Rows[dataGridView1.RowCount - 1].Height = 40;
      }
      DataGridViewCell c = dataGridView1[n % 10, n / 10];
      c.Value = ic[i];
      c.Tag = name + "-" + i;
```

```
      if (i == 0)
        c.Selected = true;
    }
  }
}
```

Листинг 29.10. Обработчик `dataGridView1.SelectionChanged`

```
private void dataGridView1_SelectionChanged(object sender, EventArgs e)
{
  if (pictureBox1.Image != null)
    pictureBox1.Image.Dispose();
  DataGridViewCell c = dataGridView1.SelectedCells[0];
  if (c.Value != null)
  {
    Text = "Icon Viewer - " + c.Tag;
    pictureBox1.Image = (c.Value as Icon).ToBitmap();
  }
  else
  {
    Text = "Icon Viewer";
    pictureBox1.Image = null;
  }
}
```

Результат. В компоненте `pictureBox1` выводится увеличенное изображение выбранного значка (для изменения размеров изображения достаточно изменить размер формы). Кроме того, в заголовке формы выводится имя выбранного значка, получаемое путем добавления к имени файла, содержащего этот значок, символа "-" (дефис) и порядкового номера значка в файле (нумерация ведется от 0, расширение в имени файла не указывается). Если выбранная ячейка не содержит значок (например, после очистки таблицы), то компонент `pictureBox1` очищается, а в заголовке формы удаляется информация об имени выбранного значка. Для формы установлен минимальный размер (свойство `MinimumSize`), позволяющий всегда отображать на экране компонент `pictureBox1` и хотя бы одну строку таблицы `dataGridView1`.

Комментарии

1. Для хранения имени значка используется свойство `Tag` той ячейки, в которой отображается значок.
2. Определить, какая ячейка таблицы является выбранной в данный момент, можно с помощью свойства-коллекции `SelectedCells` компонента `DataGridView`. Поскольку в нашей программе запрещен одновременный выбор нескольких ячеек (так как свойство `MultiSelect` равно **`False`**), свойство `SelectedCells` всегда содержит единственный элемент `SelectedCells[0]`.
3. Свойству `Image` компонента `pictureBox1` нельзя присваивать объекты типа `Icon`, поэтому предварительно приходится преобразовывать объект типа `Icon` в объект типа `Bitmap`, используя метод `ToBitmap` класса `Icon` (см. листинг 29.10).

29.5. Переключение между увеличенным и обычным изображением иконки

Определите обработчики события `Click` для компонента `pictureBox1` и события `KeyPress` для таблицы `dataGridView1` (листинг 29.11).

Листинг 29.11. Обработчики `pictureBox1.Click` и `dataGridView1.KeyPress`

```
private void pictureBox1_Click(object sender, EventArgs e)
{
  if (pictureBox1.SizeMode == PictureBoxSizeMode.Zoom)
    pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;
  else
    pictureBox1.SizeMode = PictureBoxSizeMode.Zoom;
}
private void dataGridView1_KeyPress(object sender, KeyPressEventArgs e)
{
  if (e.KeyChar == ' ')
    pictureBox1_Click(null, null);
}
```

Результат. При щелчке мышью на компоненте `pictureBox1` или нажатии клавиши <Пробел> изображение значка в компоненте `pictureBox1` изменяется с увеличенного на обычное и наоборот.

ПРИМЕЧАНИЕ

Обработчик события `KeyPress` был определен для компонента `dataGridView1`, так как это единственный компонент приложения, который может принимать фокус ввода (т. е. реагировать на события, связанные с клавиатурой). Другим вариантом является определение такого же обработчика события `KeyPress` для формы `Form1`, однако при этом необходимо дополнительно установить значение свойства `KeyPreview` формы равным **`True`**.

29.6. Сохранение значка в виде ico- или png-файла

Добавьте в форму `Form1` невизуальный компонент типа `SaveFileDialog` (этот компонент получит имя `saveFileDialog1` и будет размещен в области невизуальных компонентов под изображением формы).

Добавьте на панель инструментов `toolStrip1` два новых компонента-кнопки и измените их имена (т. е. свойство `Name`) на **`saveIco1`** и **`savePng1`**. Настройте свойства добавленных кнопок (листинг 29.12; рис. 29.3).

В метод `dataGridView1_SelectionChanged` (см. листинг 29.10) добавьте оператор

```
saveIco1.Enabled = savePng1.Enabled = c.Value != null;
```

Определите обработчики события `Click` для кнопок `saveIco1` и `savePng1` (листинг 29.13).

Рис. 29.3. Окончательный вид формы `Form1` проекта ICONVIEW

Листинг 29.12. Настройка свойств

```
saveIco1: Text = Save as &Ico-file, DisplayStyle = Text,
  AutoSize = False, Size.Width = 100
savePng1: Text = Save as &Png-file, DisplayStyle = Text,
  AutoSize = False, Size.Width = 100
```

Листинг 29.13. Обработчики `saveIco1.Click` и `savePng1.Click`

```
private void saveIco1_Click(object sender, EventArgs e)
{
  saveFileDialog1.Title = "Save icon as Ico-file";
  saveFileDialog1.DefaultExt = "ico";
  saveFileDialog1.Filter = "Ico-files (*.ico)|*.ico";
  DataGridViewCell c = dataGridView1.SelectedCells[0];
  saveFileDialog1.FileName = c.Tag.ToString();
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
  {
    FileStream fs = File.Create(saveFileDialog1.FileName);
    (c.Value as Icon).Save(fs);
    fs.Close();
  }
}
private void savePng1_Click(object sender, EventArgs e)
{
  saveFileDialog1.Title = "Save icon as Png-file";
  saveFileDialog1.DefaultExt = "png";
  saveFileDialog1.Filter = "Png-files (*.png)|*.png";
  DataGridViewCell c = dataGridView1.SelectedCells[0];
  saveFileDialog1.FileName = c.Tag.ToString();
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    pictureBox1.Image.Save(saveFileDialog1.FileName);
}
```

Результат. Кнопки **Save as Ico-file** и **Save as Png-file** позволяют сохранить текущий значок в стандартном формате для значков ICO и в сжатом графическом формате PNG соответственно. Эти кнопки доступны только в том случае, когда

текущая ячейка таблицы не является пустой. В качестве имени ico- или png-файла предлагается имя значка, отображаемое в заголовке окна программы; расширение (ico или png) добавляется автоматически. Для действий по сохранению значков в ICO- и PNG-форматах предусмотрены также клавиатурные комбинации <Alt>+<I> и <Alt>+<P> соответственно.

ПРИМЕЧАНИЕ

Для метода `Save` класса `Icon` (в отличие от одноименного метода класса `Image`) не предусмотрено варианта с параметром — именем файла; поэтому для сохранения значка в формате ICO приходится создавать файловый поток с требуемым именем, передавать его в качестве параметра метода `Save`, после чего закрывать этот поток методом `Close`.

Приведем изображение работающей программы после загрузки значков из файла Moricons.dll, находящегося в подкаталоге System32 каталога Windows (рис. 29.4).

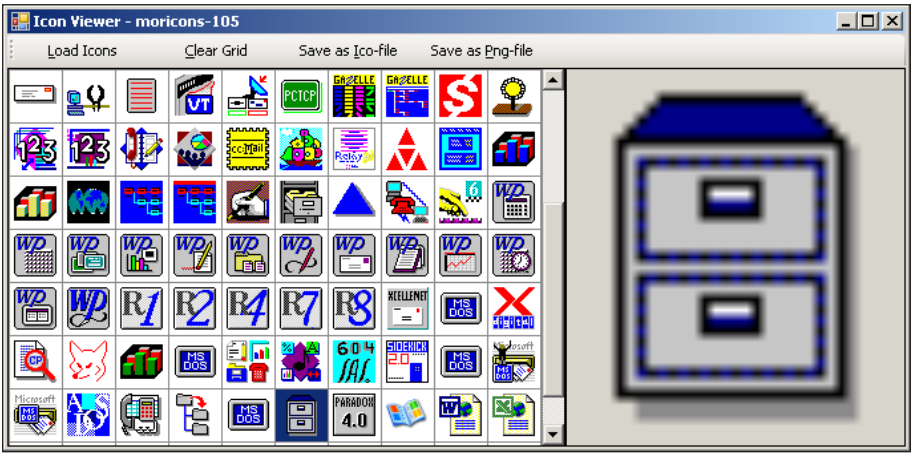


Рис. 29.4. Вид работающего приложения ICONVIEW

Глава 30

Приложение с заставкой: проект TRIGFUNC

Проект TRIGFUNC демонстрирует дополнительные возможности компонента-таблицы `DataGridView`, связанные с настройкой свойств ее столбцов. В нем также описываются приемы, связанные с обработкой параметров командной строки, отображением на экране окна-заставки с особыми свойствами и досрочным завершением программы. Рассматривается метод `Sleep` класса `Thread`, метод `DoEvents` класса `Application` и компонент `ProgressBar` (индикатор хода процесса).

30.1. Формирование таблицы значений тригонометрических функций

После создания проекта TRIGFUNC разместите в форме `Form1` компонент-таблицу `dataGridView1` и настройте свойства формы и добавленного компонента (листинг 30.1).

Кроме настройки свойств таблицы `dataGridView1` "в целом", необходимо настроить свойства ее столбцов. Для этого выделите компонент `dataGridView1`, выберите в окне **Properties** свойство `Columns` и нажмите расположенную рядом с ним кнопку с многоточием [...]. На экране появится диалоговое окно **Edit Columns**, позволяющее создавать столбцы таблицы, задавать их порядок и настраивать их свойства. Нажмите в этом окне кнопку **Add**, в появившемся окне **Add Column** введите в поле **Header text** заголовок первого столбца: **`x`** (прочие настройки менять не требуется) и нажмите кнопку **Add**. В левой части окна **Edit Columns** появится строка, соответствующая созданному столбцу, а в правой — таблица, позволяющая настраивать его свойства (структура данной таблицы подобна структуре окна **Property**), причем окно **Add Column** останется на экране. Аналогичными действиями добавьте в таблицу `dataGridView1` еще четыре столбца, указав для них следующие заголовки: **`Sin(x*pi)`**, **`Cos(x*pi)`**, **`Tg(x*pi)`**, **`Ctg(x*pi)`**. После этого закройте окно **Add Column**, нажав кнопку **Close**, и окно **Edit Columns**, нажав кнопку **OK**. В результате в компоненте

dataGridView1 будут изображены заголовки созданных столбцов (рис. 30.1). См. также комментарии 1 и 2.

Определите обработчик события Load для формы Form1 (листинг 30.2).

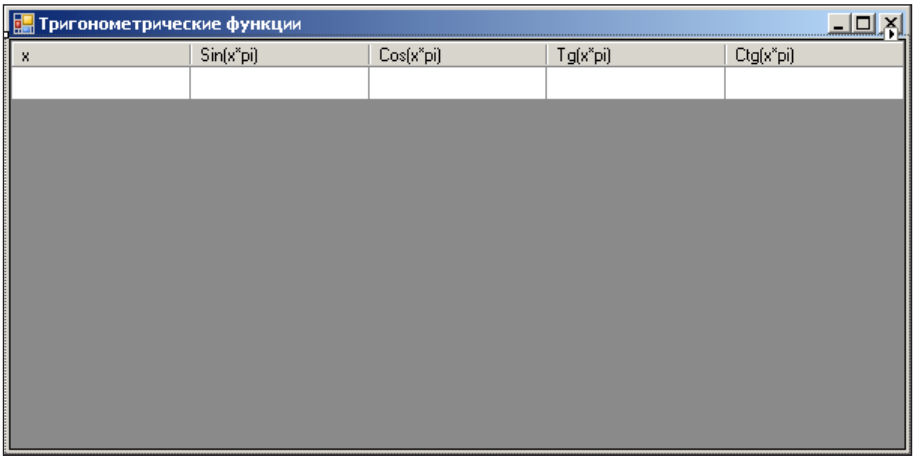


Рис. 30.1. Вид формы Form1 для проекта TRIGFUNC на начальном этапе разработки

Листинг 30.1. Настройка свойств

```
Form1: Text = Тригонометрические функции, StartPosition = CenterScreen
dataGridView1: Dock = Fill, AllowUserToResizeColumns = False,
  AllowUserToResizeRows = False, AutoSizeColumnsMode = Fill,
  RowHeadersVisible = False, ReadOnly = True
```

Листинг 30.2. Обработчик Form1.Load

```
private void Form1_Load(object sender, EventArgs e)
{
  int n = 7,
    nMax = 100001;
  string[] args = Environment.GetCommandLineArgs();
  if (args.Length > 1)
    try
    {
      int n0 = int.Parse(args[1]);
      if (n0 < 2 || n0 > nMax)
```

```
        throw new Exception();
      else
        n = n0;
    }
    catch
    {
      string s = string.Format("Неверный параметр: {0}\n" +
        "Допустимые значения: от 2 до {1}", args[1], nMax);
      MessageBox.Show(s, "Ошибка", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
      Close();
      return;
    }
  double step = 1.0 / (n - 1);
  // уменьшаем вдвое ширину первого столбца:
  dataGridView1.Columns[0].Width /= 2;
  dataGridView1.RowCount = n;
  for (int i = 0; i < n; i++)
  {
    double x = i * step,
      sx = Math.Sin(Math.PI * x),
      cx = Math.Cos(Math.PI * x);
    dataGridView1[0, i].Value = x.ToString("f5");
    dataGridView1[1, i].Value = sx.ToString();
    dataGridView1[2, i].Value = cx.ToString();
    dataGridView1[3, i].Value = (sx / cx).ToString();
    dataGridView1[4, i].Value = (cx / sx).ToString();
  }
}
```

Результат. После запуска программа заполняет таблицу `dataGridView1` значениями тригонометрических функций с аргументами от 0 до π радианов на сетке из `n` равноотстоящих точек. Число точек `n` можно указать в качестве параметра командной строки; допускаются значения от 2 до 100001. При запуске программы из среды Visual C# параметр командной строки можно указать в поле **Command line arguments** группы настроек **Debug** на вкладке свойств проекта (напомним, что эту вкладку можно отобразить в редакторе командой

Project | *Имя проекта* Properties). Если параметр указан неверно, то выдается сообщение об ошибке и программа немедленно завершает работу. Если параметр не указан, то количество точек полагается равным 7. См. также комментарии 3—8.

Комментарии

1. Обратите внимание на то, что при добавлении нового столбца можно указать его тип, используя выпадающий список **Type**. Тип столбца определяет, какой тип будет использован по умолчанию для всех ячеек данного столбца (возможные типы столбцов таблицы перечислены в комментарии 1 в *разд. 29.2*). Помимо типа столбца и текста (`HeaderText`), отображаемого в его заголовке, в окне **Add Column** можно указать имя столбца (`Name`) — идентификатор, подобный именам компонентов, размещенных в форме (по умолчанию предлагаются имена `Column1`, `Column2` и т. д.). Кроме того, при создании нового столбца можно сразу определить три его свойства логического типа: `Visible` (значение по умолчанию **`True`**), `ReadOnly` и `Frozen` (значения по умолчанию **`False`**). В пояснении нуждается только последнее свойство: если `Frozen` равно **`True`**, то данный столбец и все предшествующие ему столбцы "замораживаются", т. е. перестают перемещаться на экране при горизонтальной прокрутке таблицы `DataGridView`. Все указанные свойства, а также ряд других, можно определить и после создания столбца, с помощью таблицы свойств в окне **Edit Columns**. Для вывода на экран диалоговых окон **Add Colum**n и **Edit Columns** необязательно использовать свойство `Columns` в окне **Properties**; достаточно выполнить щелчок мышью на значке ▶, который отображается в правом верхнем углу компонента `DataGridView` при его выделении (см. рис. 30.1), и в появившейся вспомогательной панели **DataGridView Tasks** выбрать пункт **Add Column...** или **Edit Columns...**.
2. Определить столбцы таблицы `DataGridView` и настроить их свойства можно и без применения визуальных средств. Проще всего для этого воспользоваться одним из двух вариантов метода `Add` свойства `Columns` компонента `DataGridView`. Первый вариант метода `Add` имеет два строковых параметра `columnName` и `headerText` и позволяет создавать столбец с текстовыми данными, т. е. столбец типа `DataGridViewTextBoxColumn`. Второй, универсальный вариант позволяет добавлять к таблице объект-столбец любого из шести возможных типов (см. комментарий 1 в *разд. 29.2*); добавляемый столбец указывается в качестве единственного параметра этого варианта метода `Add` (параметр имеет тип `DataGridViewColumn`). Метод `Add` возвращает индекс созданного столбца. В нашем случае, с учетом того, что все столбцы имеют

тип `DataGridViewTextBoxColumn`, а их имена в дальнейшем использоваться не будут, вместо описанных выше действий по настройке столбцов можно было бы просто поместить в конструктор класса `Form1` следующие пять операторов:

```
dataGridView1.Columns.Add("", "x");
dataGridView1.Columns.Add("", "Sin(x*pi)");
dataGridView1.Columns.Add("", "Cos(x*pi)");
dataGridView1.Columns.Add("", "Tg(x*pi)");
dataGridView1.Columns.Add("", "Ctg(x*pi)");
```

3. Для доступа к параметрам командной строки был использован метод `GetCommandLineArgs` класса `Environment`, возвращающий строковый массив. Первый элемент этого массива (с индексом 0) содержит полное имя исполняемого файла приложения, а начиная со следующего элемента, в массиве размещаются параметры командной строки. Напомним, что параметры командной строки ранее рассматривались в *разд. 2.3* при разработке консольного приложения.

4. Указанный при запуске программы параметр командной строки может оказаться недопустимым по двум причинам: во-первых, он может быть строкой, которую нельзя преобразовать к целому числу, во-вторых, даже если подобное преобразование возможно, полученное число может лежать вне допустимого диапазона. Вывод сообщения о любой из возможных ошибок выполняется в разделе `catch` `try`-блока (см. листинг 30.2). В начале этого `try`-блока делается попытка преобразовать параметр командной строки в целое число. При невозможности подобного преобразования возбуждается исключение, которое сразу передает управление в раздел `catch`. Если же преобразование прошло успешно, однако число находится вне допустимого диапазона, то для перехода в раздел `catch` программа *явным образом* генерирует исключение, используя для этого оператор `throw` (тип исключения в данном случае роли не играет, поэтому возбуждается исключение типа `Exception` — общего предка всех классов-исключений).

5. Для немедленного завершения программы достаточно вызвать метод `Close` главной формы (в нашей программе этот метод вызывается в разделе `catch` `try`-блока). После вызова метода `Close` следует сразу выйти из текущего обработчика, используя оператор `return`.

6. Благодаря значению **`Fill`** свойства `AutoSizeColumnsMode` компонента `dataGridView1`, столбцы "растягиваются" по ширине компонента. По умолчанию ширина всех столбцов является одинаковой, однако для любого столбца ее можно изменить; так, в нашей программе была вдвое уменьшена

ширина первого столбца (соответствующий оператор в листинге 30.2 снабжен комментарием).

7. При выполнении вычислений с типом `double` можно не беспокоиться о возможном переполнении (см. комментарий 2 в *разд. 3.1*). В частности, при вычислении значения функции `Ctg` в нуле ошибки не произойдет, а в соответствующей ячейке таблицы будет выведено значение **`бесконечность`**.
8. Для преобразования полученных значений функций к их строковому представлению в нашей программе используется метод `ToString` без параметров, возвращающий наиболее краткое из возможных представлений данного числа (так называемый *общий формат* — general). Любой другой вариант форматирования надо явным образом указать в качестве параметра метода `ToString`. Для столбца аргументов `x` мы используем формат `f5` — формат с фиксированным числом дробных знаков (в данном случае — с пятью знаками после запятой).

30.2. Отображение окна-заставки при загрузке программы

Добавьте к проекту новую форму `Form2` и разместите в ней метку `label1`. Настройте свойства формы `Form2` и метки `label1` (листинг 30.3). Для настройки указанных свойств (`Name`, `Size` и `Bold`) свойства `Font` метки `label1` удобно вывести на экран диалоговое окно **Шрифт**, нажав кнопку с многоточием  рядом со свойством `Font` в окне **Properties**. С помощью окна **Шрифт** можно настроить все свойства шрифта и, кроме того, увидеть образец шрифта с текущими настройками.

После настройки свойства `Font` компонента `label1` измените размер формы `Form2` так, чтобы текст метки размещался в двух строках (рис. 30.2).

> ***ПРИМЕЧАНИЕ***
>
> Напомним, что для выбора формы в ситуации, когда выбран один из ее компонентов, достаточно нажать клавишу <Esc>. Еще один быстрый способ выбора формы — щелчок мышью на ее заголовке — в данном случае неприменим, так как выполненная ранее настройка свойств формы (см. листинг 30.3) привела к тому, что заголовок в ней отсутствует.

В описание класса `Form1` в файле Form1.cs добавьте новое поле:

```
private Form2 form2 = new Form2();
```

Дополните конструктор класса `Form1` (листинг 30.4).

Рис. 30.2. Вид формы Form2 для проекта TRIGFUNC на начальном этапе разработки

В конец метода Form1_Load, описанного в листинге 30.2, добавьте оператор

```
form2.Hide();
```

Подчеркнем, что этот оператор должен располагаться *после* цикла for, в котором происходит заполнение таблицы dataGridView1 значениями тригонометрических функций.

Листинг 30.3. Настройка свойств формы Form2 и ее компонентов

```
Form2: Text = пустая строка, ControlBox = False,
  FormBorderStyle = FixedSingle, Opacity = 80%, ShowInTaskbar = False,
  StartPosition = CenterScreen, UseWaitCursor = True
label1: Text = Тригонометрические функции, AutoSize = False, Dock = Fill,
  TextAlign = MiddleCenter, Font.Name = Times New Roman, Font.Size = 32,
  Font.Bold = True
```

Листинг 30.4. Новый вариант конструктора формы Form1

```
public Form1()
{
  InitializeComponent();
  AddOwnedForm(form2);
  form2.Show();
  Application.DoEvents();
}
```

Результат. В ходе загрузки главной формы Form1 и заполнения таблицы dataGridView1 данными на экране отображается окно-заставка Form2,

свидетельствующая о том, что программа запущена и в настоящий момент выполняет инициализирующие действия. Заставка не имеет заголовка и является полупрозрачной; ее нельзя перемещать по экрану. Курсор мыши на заставке принимает вид песочных часов. При отображении главной формы заставка исчезает.

Недочет. Если объем вычислений, необходимый для заполнения таблицы, невелик (например, при использовании числа точек `n = 7`, заданного по умолчанию), то рассмотреть заставку не удается из-за краткого времени ее отображения.

Исправление. В методе `Form1_Load` перед добавленным ранее оператором

```
form2.Hide();
```

вставьте еще один оператор:

```
System.Threading.Thread.Sleep(1000);
```

Результат. После завершения инициализирующих действий программа приостанавливает работу на 1 секунду, в течение которой окно-заставка остается на экране.

Комментарии

1. За режим прозрачности формы отвечает ее свойство `Opacity`. По умолчанию оно равно `100%`, что соответствует режиму полной непрозрачности. Режим частичной прозрачности удобен для диалоговых окон, если при их отображении на экране желательно видеть расположенный под ними текст основных окон (заметим, что подобные окна используются в среде Visual C# при выводе сообщений о возникших исключениях (в режиме **Debug**). В нашем случае мы использовали полупрозрачное окно-заставку только для того, чтобы придать ему более оригинальный вид.
2. При отсутствии оператора `Application.DoEvents()` (см. листинг 30.4) вместо окна-заставки на экране изображалась бы пустая прямоугольная область. Вызов статического метода `DoEvents` класса `Application` обеспечивает немедленную обработку всех событий, возникших к данному моменту, в частности, событий, связанных с перерисовкой компонентов на экране. Точнее, метод `DoEvents` выполняет обработку всех *сообщений Windows*, содержащихся в текущий момент в очереди сообщений для данного приложения (напомним, что в .NET-приложениях сообщения Windows преобразуются в *события* для тех или иных компонентов).
3. Метод `Sleep(ms)` класса `Thread` из пространства имен `System.Threading` приостанавливает выполнение программы на указанное число миллисекунд `ms`.

30.3. Использование окна-заставки в качестве информационного окна

Разместите в форме `Form2` кнопку `button1` и настройте свойства формы `Form2` и добавленного компонента (листинг 30.5; рис. 30.3).

Разместите в форме `Form1` компонент-меню `menuStrip1`. Используя конструктор меню *(см. разд. 18.1)*, создайте в компоненте `menuStrip1` пункт меню первого уровня с текстом **`&About...`** (рис. 30.4) и с помощью окна **Properties** измените имя этого пункта (т. е. свойство `Name`) на **`about1`**. Определите обработчик события `Click` для созданного пункта меню (листинг 30.6).

В конец метода `Form1_Load` добавьте два оператора:

```
form2.UseWaitCursor = false;
form2.Controls["button1"].Visible = true;
```

Рис. 30.3. Вид формы `Form2` для проекта TRIGFUNC на промежуточном этапе разработки

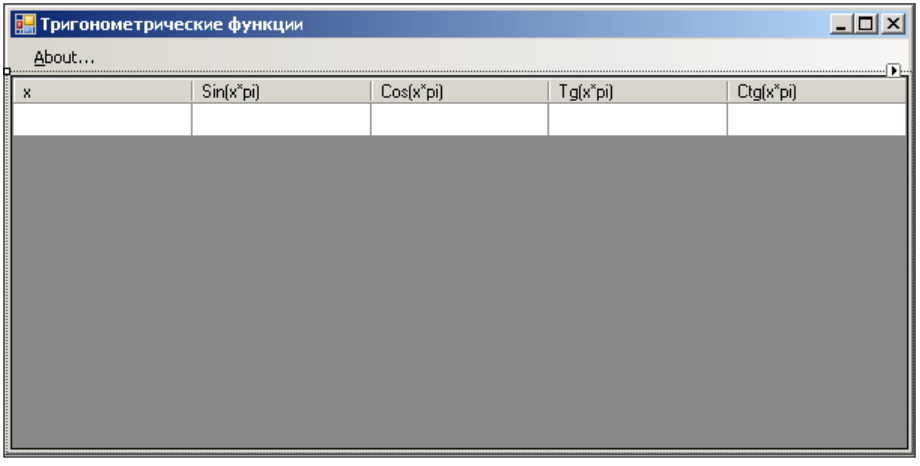


Рис. 30.4. Окончательный вид формы `Form1` для проекта TRIGFUNC

Листинг 30.5. Настройка свойств формы Form2 и ее компонентов

```
Form2: AcceptButton = button1
button1: Text = OK, DialogResult = OK, Visible = False
```

Листинг 30.6. Обработчик about1.Click

```
private void about1_Click(object sender, EventArgs e)
{
  form2.ShowDialog();
}
```

Результат. Теперь окно-заставку можно выводить на экран после загрузки главной формы, используя команду меню **About...**. Для закрытия окна-заставки в этом случае предназначена кнопка **OK**. При выводе окна-заставки в ходе загрузки главной формы кнопка **OK** в нем не отображается.

30.4. Вывод в окне-заставке информации о ходе загрузки программы

Разместите в форме Form2 компонент типа ProgressBar (он получит имя progressBar1) и настройте его свойства (листинг 30.7; рис. 30.5).

Измените метод Form1_Load формы Form1 (листинг 30.8).

Рис. 30.5. Окончательный вид формы Form2 для проекта TRIGFUNC

Листинг 30.7. Настройка свойств

```
progressBar1: Dock = Bottom, Modifiers = Internal, Style = Continuous
```

Листинг 30.8. Новый вариант метода Form1_Load

```
private void Form1_Load(object sender, EventArgs e)
{
  int n = 7,
    nMax = 100001;
  string[] args = Environment.GetCommandLineArgs();
  if (args.Length > 1)
    try
    {
      int n0 = int.Parse(args[1]);
      if (n0 < 2 || n0 > nMax)
        throw new Exception();
      else
        n = n0;
    }
    catch
    {
      string s = string.Format("Неверный параметр: {0}\n" +
        "Допустимые значения: от 2 до {1}", args[1], nMax);
      MessageBox.Show(s, "Ошибка", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
      Close();
      return;
    }
  double step = 1.0 / (n - 1);
  dataGridView1.Columns[0].Width /= 2;
  dataGridView1.RowCount = n;
  for (int i = 0; i < n; i++)
  {
    double x = i * step,
      sx = Math.Sin(Math.PI * x),
      cx = Math.Cos(Math.PI * x);
    dataGridView1[0, i].Value = x.ToString("f5");
    dataGridView1[1, i].Value = sx.ToString();
    dataGridView1[2, i].Value = cx.ToString();
```

```
    dataGridView1[3, i].Value = (sx / cx).ToString();
    dataGridView1[4, i].Value = (cx / sx).ToString();
    form2.progressBar1.Value = 100 * i / (n - 1);
  }
  System.Threading.Thread.Sleep(1000);
  form2.Hide();
  form2.UseWaitCursor = false;
  form2.Controls["button1"].Visible = true;
  form2.progressBar1.Visible = false;
}
```

Результат. В ходе загрузки главной формы в окне-заставке выводится графическая информация о том, какая часть инициализирующих действий уже выполнена (для этого используется компонент `progressBar1`). При последующих вызовах окна-заставки командой **About...** компонент `progressBar1` в нем не отображается.

Комментарии

1. Компонент `ProgressBar` *(индикатор хода процесса)* позволяет отображать информацию о ходе выполнения некоторого процесса в трех режимах, определяемых свойством `Style`: использованная в нашей программе индикация в виде сплошной полосы (**`Continuous`**), индикация в виде набора прямоугольных блоков (**`Blocks`**, значение по умолчанию) и индикация в виде бегущих маркеров (**`Marquee`**). Последний режим удобно использовать в ситуации, когда надо указать на выполнение некоторых продолжительных действий, однако сложно определить, какая часть действий уже выполнена. В режиме `Marquee` не требуется никакой дополнительной настройки компонента `ProgressBar`; отметим лишь, что с помощью свойства `MarqueeAnimationSpeed` можно задать скорость перемещения маркеров. В двух других режимах вид индикатора зависит от значений трех целочисленных свойств: `Value`, `Minimum` (по умолчанию значения обоих свойств равны `0`) и `Maximum` (значение по умолчанию `100`). Значение свойства `Value` лежит в диапазоне от `Minimum` до `Maximum` и определяет, какая часть индикатора будет выделена. Хотя в компоненте `ProgressBar` для изменения значения `Value` предусмотрены методы `Increment` и `PerformStep`, в этих методах нет особой необходимости, поскольку свойство `Value` доступно как для чтения, так и для записи.
2. Положив значение свойства `Modifiers` компонента `progressBar1` равным **`Internal`**, мы получили возможность прямого доступа к этому компоненту

из методов другой формы (а именно формы Form1). При этом, в частности, отпала необходимость в использовании свойства-коллекции Controls (которое применялось в *разд. 30.3* для доступа из формы Form1 к компоненту button1 формы Form2).

30.5. Досрочное завершение программы

Определите обработчик события KeyDown для формы Form2 (листинг 30.9).

В методе Form1_Load формы Form1 (его последний вариант приведен в листинге 30.8) в начало цикла for вставьте новый фрагмент:

```
Application.DoEvents();
if (!form2.Visible)
{
  Close();
  return;
}
```

Листинг 30.9. Обработчик Form2.KeyDown

```
private void Form2_KeyDown(object sender, KeyEventArgs e)
{
  if (e.KeyCode == Keys.Escape)
    Hide();
}
```

Результат. Теперь выполнение программы можно прервать в процессе проведения начальных вычислений; для этого достаточно нажать клавишу <Esc>. При этом окно-заставка исчезает с экрана, и программа немедленно завершает работу.

Комментарии

1. Следует обратить внимание на вызов метода DoEvents перед проверкой того, остается ли форма Form2 видимой на экране. Метод DoEvents позволяет обработать события, возникшие уже после начала выполнения метода Form1_Load (а именно таким событием будет нажатие клавиши <Esc>). При отсутствии метода DoEvents все подобные события будут обработаны только после завершения текущего метода, т. е. после того, как все действия, связанные с инициализацией исходных данных, будут завершены, что в нашем случае является неприемлемым.

2. При отображении окна-заставки с помощью команды **About...** нажатие на клавишу <Esc> не приводит к его закрытию. Это связано с тем, что в данном случае события, связанные с клавиатурой, поступают непосредственно к активному компоненту `button1`, минуя форму `Form2`, для которой определен обработчик. Для того чтобы клавиша <Esc> обеспечивала закрытие окна и в этом случае, достаточно положить свойство `KeyPreview` формы `Form2` равным **`True`**. Если внести в программу такое исправление, то окно, вызванное на экран командой **About...**, можно будет закрыть либо клавишей <Enter>, либо клавишей <Esc>.

30.6. Возможность перемещения окна-заставки

В описание класса `Form2` добавьте поле

```
private Point p;
```

Определите обработчики событий `MouseDown` и `MouseMove` для компонента `label1` формы `Form2` (листинг 30.10).

Листинг 30.10. Обработчики `label1.MouseDown` и `label1.MouseMove` (форма `Form2`)

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
}
private void label1_MouseMove(object sender, MouseEventArgs e)
{
  if (Control.MouseButtons == MouseButtons.Left)
    Location += new Size(e.X - p.X, e.Y - p.Y);
}
```

Результат. Окно-заставку теперь можно перемещать по экрану с помощью мыши; для этого надо установить курсор мыши в любой позиции окна, нажать левую кнопку мыши и, не отпуская ее, переместить мышь в требуемую позицию экрана.

> ***ПРИМЕЧАНИЕ***
>
> Ранее подобный способ перетаскивания был использован в *главе 10*. Этот способ является наиболее естественным для тех окон, которые не содержат заголовка.

Недочет. Окно-заставку можно перетаскивать и при его первом отображении на экране (во время инициализации данных), что обычно является нежелательным.

Исправление. Выделите компонент `label1` и в окне **Properties** *очистите* поле, связанное с событием `MouseMove` этого компонента. В описании метода `label1_MouseMove` замените модификатор `private` на **`internal`**. В файле Form1.cs в конце метода `Form1_Load` добавьте следующий оператор:

```
form2.Controls["label1"].MouseMove += form2.label1_MouseMove;
```

Результат. Теперь перетаскивание окна-заставки возможно только при его отображении на экране командой **About...**. В ходе начальной загрузки программы окно-заставку перетаскивать нельзя, поскольку к этому времени метод `label1_MouseMove` еще не связан с событием `MouseMove` компонента `label1`.

Приведем изображение работающей программы после вызова окна-заставки командой **About...** в случае, когда число точек `n` равно 9 (рис. 30.6).

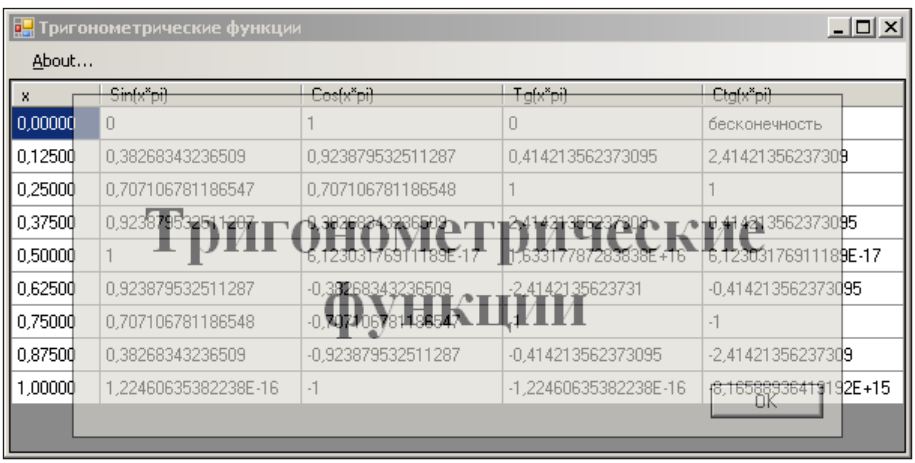


Рис. 30.6. Вид работающего приложения TRIGFUNC

Глава 31

Обработка наборов данных: проект STUD1

Проект STUD1 посвящен стандартным средствам .NET, связанным с обработкой наборов данных, включая их привязку к компонентам формы (data binding). В нем создается класс `Student` с описанием структуры данных, рассматривается компонент `BindingSource` и описываются действия, необходимые для связывания с помощью данного компонента созданной структуры с таблицей типа `DataGridView`. Также описываются способы контроля правильности вводимых данных на уровне отдельной ячейки таблицы и строки в целом, демонстрируется использование столбцов таблицы различных типов (столбцы с ячейками-флажками и ячейками — выпадающими списками) и рассматриваются простейшие средства XML-сериализации наборов данных. Описывается компонент `BindingNavigator`, предоставляющий дополнительные средства навигации и редактирования набора данных, и приводится пример автоматизации действий при добавлении нового элемента в набор данных. Дополнительно описывается механизм автогенерации кода, реализованный в среде Visual C#, и связанные с ним шаблоны (code snippets).

31.1. Определение класса с описанием структуры данных и связывание этой структуры с таблицей в режиме дизайна

После создания проекта STUD1 добавьте в форму `Form1` таблицу `dataGridView1`, а также невизуальный компонент типа `BindingSource` (этот компонент получит имя `bindingSource1` и будет размещен ниже формы, в области невизуальных компонентов). Настройте свойства формы и компонента `dataGridView1` (листинг 31.1; для задания свойства `DataSource` компонента `dataGridView1` необходимо нажать кнопку разворачивания списка справа от этого свойства

в окне **Properties** и в появившемся вспомогательном окне выбрать вариант **bindingSource1**).

Листинг 31.1. Настройка свойств

```
Form1: Text = Students, StartPosition = CenterScreen
dataGridView1: Dock = Fill, AutoSizeColumnsMode = Fill,
  DataSource = bindingSource1
```

Для связывания таблицы с набором данных (с возможностью его последующего просмотра, редактирования и сохранения) в .NET Framework предусмотрены удобные механизмы, описываемые далее в этом разделе. Однако для реализации этих механизмов необходимо оформить набор данных в виде вспомогательного класса, в котором каждое *поле* элемента данных должно быть представлено в виде *свойства* класса. Кроме того, необходимо, чтобы этот вспомогательный класс имел конструктор без параметров и был снабжен модификатором доступа `public`.

Создадим класс `Student` со следующими свойствами:

- `Id` (порядковый номер студента, тип `int`),
- `Surname` (фамилия студента, тип `string`),
- `Gender` (пол, тип `string`),
- `BirthDate` (дата рождения, тип `DateTime`),
- `Course`, `Group` (номер курса и группы, оба типа `int`),
- `Scholarship` (получает ли студент стипендию, тип `bool`).

Опишем этот класс в новом файле Student.cs. Напомним *(см. разд. 29.1)*, что для создания файла с описанием нового класса `Student` следует выполнить команду меню **Project | Add Class...**, в появившемся диалоговом окне заменить имя, предлагаемое по умолчанию, на Student.cs, после чего нажать клавишу <Enter> или кнопку **Add**.

Нам потребуется добавить в описание созданного класса `Student` целый ряд свойств и связанных с ними закрытых полей класса. Ускорить выполнение этих действий можно с помощью механизма *автогенерации кода*, который мы опишем на примере добавления к классу свойства `Id` (см. также комментарий).

В файле Student.cs переместите клавиатурный курсор внутрь блока `{}` с описанием класса `Student`, наберите текст **prop** и нажмите клавишу <Tab>. В результате будет автоматически создана заготовка для определения нового открытого свойства и связанного с ним закрытого поля (листинг 31.2; если после первого нажатия клавиши <Tab> заготовка не появится, то нажмите эту клавишу еще раз).

Листинг 31.2. Текст заготовки для определения нового свойства

```
private int myVar;

public int MyProperty
{
  get { return myVar; }
  set { myVar = value; }
}
```

В полученной заготовке достаточно изменить три выделенных фрагмента (в листинге 31.2 эти фрагменты заключены в рамку). Так как свойство `Id` должно иметь тип `int`, первый фрагмент изменять не требуется, поэтому еще раз нажмите <Tab> для выделения имени `myVar` и введите вместо него имя закрытого поля: **`id`** (в качестве имен закрытых полей, связанных со свойствами, мы будем использовать имена соответствующих свойств, набранных со строчной буквы).

Обратите внимание на то, что при очередном нажатии клавиши <Tab> в строках `get` и `set` имя `myVar` будет также заменено на `id`. Осталось ввести имя открытого свойства **`Id`**, после чего созданный фрагмент примет окончательный вид (листинг 31.3). Для завершения редактирования свойства `Id` достаточно нажать клавишу <Enter>; в результате клавиатурный курсор переместится на позицию за описанием этого свойства.

Листинг 31.3. Окончательный вариант описания свойства Id

```
private int id;

public int Id
{
  get { return id; }
  set { id = value; }
}
```

Аналогичными действиями, используя механизм автогенерации кода, добавьте в класс `Student` оставшиеся свойства и связанные с ними поля. Отметим, что с классом `Student` будет автоматически связан конструктор без параметров, обеспечивающий инициализацию всех полей нулевыми значениями соответствующего типа (описывать такой конструктор, называемый *конструктором по умолчанию*, не требуется).

ПРИМЕЧАНИЕ

Конструктор по умолчанию автоматически добавляется к классу только в случае, если в описании класса отсутствует явное определение хотя бы одного конструктора.

Если при инициализации какого-либо поля следует использовать значение, отличное от нулевого, его можно просто указать после описания этого поля. Сделаем это для поля `gender`, инициализировав его строкой, содержащей заглавную русскую букву М:

```
private string gender = "М";
```

Осталось добавить к описанию класса модификатор **`public`**. В результате текст файла Student.cs примет вид, приведенный в листинге 31.4.

Листинг 31.4. Текст файла Student.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace STUD1
{
  public class Student
  {
    private int id;
    public int Id
    {
      get { return id; }
      set { id = value; }
    }
    private string surname;
    public string Surname
    {
      get { return surname; }
      set { surname = value; }
    }
    private string gender = "М";
    public string Gender
    {
```

```
      get { return gender; }
      set { gender = value; }
    }
    private DateTime birthDate;
    public DateTime BirthDate
    {
      get { return birthDate; }
      set { birthDate = value; }
    }
    private int course;
    public int Course
    {
      get { return course; }
      set { course = value; }
    }
    private int group;
    public int Group
    {
      get { return group; }
      set { group = value; }
    }
    private bool scholarship;
    public bool Scholarship
    {
      get { return scholarship; }
      set { scholarship = value; }
    }
  }
}
```

После завершения описания класса `Student` приложение STUD1 необходимо откомпилировать для того, чтобы информацию о созданном классе можно было использовать при описываемой далее настройке свойства `DataSource` компонента `bindingSource1`.

В качестве значения свойства `DataSource` компонента `bindingSource1` нам надо указать имя только что определенного класса `Student`. Однако, в отличие от большинства свойств, настраиваемых в окне **Properties**, свойство `DataSource`

нельзя ввести в виде обычной текстовой строки. Для указания этого свойства надо выполнить следующие действия:

1. Нажмите кнопку разворачивания списка справа от свойства `DataSource` в окне **Properties**.
2. В появившемся окне щелкните мышью на тексте **Add Project Data Source...**.
3. В новом окне **Data Source Configuration Wizard** выберите вариант **Object** и нажмите кнопку **Next**.
4. Разверните появившийся древовидный список **STUD1**, в этом списке разверните пункт **{} STUD1**, в появившемся списке выделите пункт **Student**, после чего нажмите кнопку **Finish**.

После выполнения описанных действий все вспомогательные окна будут закрыты, а свойство `DataSource` компонента `bindingSource1` получит значение **`STUD1.Student`**. При этом в таблицу `dataGridView1` будут автоматически добавлены столбцы, соответствующие всем открытым свойствам класса `Student` (порядок следования столбцов соответствует порядку описания свойств в классе `Student`). Каждый столбец будет иметь заголовок, совпадающий с именем соответствующего свойства (рис. 31.1). Все столбцы, кроме последнего (**Scholarship**), будут иметь тип `DataGridViewTextBoxColumn`; последний столбец получит тип `DataGridViewCheckBoxColumn`, поскольку связанное с ним свойство `Scholarship` имеет тип `bool`.

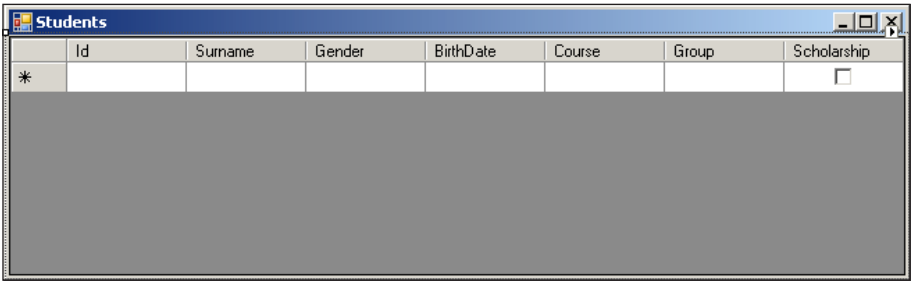


Рис. 31.1. Вид формы `Form1` для проекта STUD1 на начальном этапе разработки

Результат. После запуска полученной программы в таблицу `dataGridView1` можно вводить данные о студентах. Для ввода нового элемента данных предусмотрена последняя строка (строка NewRow), помеченная символом ***** (звездочка). Сразу после начала ввода нового элемента для него выделяется новая строка, а строка NewRow перемещается ниже. После создания новой строки в нее заносятся значения полей по умолчанию, которые определяются

по соответствующим значениям свойств класса `Student` (в частности, в столбцы **Id**, **Course**, **Group** заносятся числа `0`, а в столбец **Gender** — строка `"М"`).

При вводе данных в одну из ячеек соответствующая строка таблицы переходит в *режим редактирования*, признаком которого является изображение карандаша в заголовке этой строки. Для перехода в режим редактирования текущей ячейки можно также нажать клавишу <F2>. Для выхода из режима редактирования достаточно перейти на другую ячейку (например, с помощью клавиши <Tab>).

Недочет. При попытке ввода значения, которое не является допустимым для текущего поля (например, при вводе буквы в любое числовое поле), на экран выводится диалоговое окно **DataGridView Default Error Dialog** с сообщением об ошибке. Это сообщение содержит информацию на английском языке, которая является достаточно полезной для программиста-разработчика, но может сильно испугать обычного пользователя. Следует также отметить, что в нашем варианте программы возможен ввод формально правильных, но бессмысленных значений, например, какой-либо цифры в поле **Gender**, пустой строки в поле **Surname** или слишком поздней/ранней даты в поле **BirthDate**. Способы исправления отмеченного недочета будут приведены в *разд. 31.2* и *31.3*.

ПРИМЕЧАНИЕ

После закрытия окна с информацией об ошибке имеются два варианта продолжения: можно либо исправить введенное неверно значение, либо нажать клавишу <Esc>, в результате чего будет восстановлено прежнее (правильное) значение.

Приведем вид окна программы после ввода данных о трех студентах (рис. 31.2). Таблица находится в режиме редактирования данных о третьем студенте.

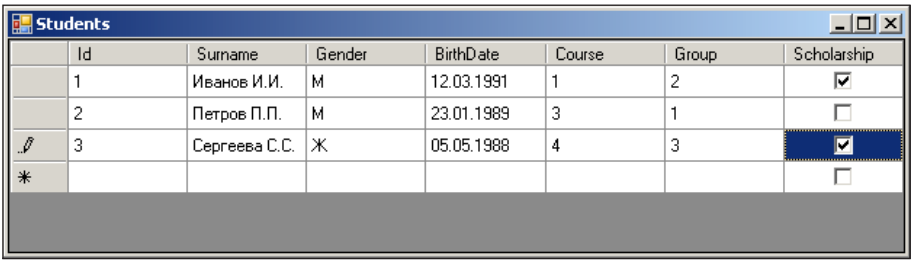


Рис. 31.2. Вид работающего приложения STUD1 на начальном этапе разработки

Комментарий

В редакторе среды Visual Studio .NET предусмотрено около 40 шаблонов для автогенерации кода (code snippets). Для генерации соответствующего кода надо

сразу после ввода шаблона нажать клавишу <Tab>. Использованный нами шаблон **prop** — один из самых удобных, так как он позволяет единственный раз ввести данные, которые требуется указывать в нескольких местах определяемого свойства. Отметим ряд других шаблонов, обладающих указанным полезным качеством: **propg** — описание свойства, доступного только для чтения; **indexer** — описание свойства-*индексатора*; **for** — шаблон для цикла `for` с возрастающим параметром; **forr** — шаблон для цикла `for` с убывающим параметром. Многие шаблоны просто генерируют небольшие фрагменты кода, состоящие из нескольких элементов. Таковы шаблоны **class**, **ctor** (для создания конструктора), **do** (для цикла `do–while`), **else**, **enum**, **foreach**, **if**, **interface**, **namespace**, **struct**, **switch**, **try**, **tryf** (для блока `try–finally`), **using**, **while** и некоторые другие. Имеются шаблоны, позволяющие создавать заготовки для описания достаточно сложных объектов. К ним относятся **attribute** (описание атрибута), **equals** (описание метода `Equals`), **exception** (описание класса-исключения) и ряд других. Наконец, отметим два шаблона, предназначенных для быстрого набора часто используемых методов с длинными именами: **cw** — вызов метода `Concole.WriteLine`, **mbox** — вызов метода `MessageBox.Show`.

31.2. Проверка правильности данных на уровне ячейки таблицы

Определите обработчик события `CellValidating` для таблицы `dataGridView1` (листинг 31.5).

Листинг 31.5. Обработчик `dataGridView1.CellValidating`

```
private void dataGridView1_CellValidating(object sender,
  DataGridViewCellValidatingEventArgs e)
{
  string err = "",
    s = e.FormattedValue.ToString();
  switch (e.ColumnIndex)
  {
    case 0:  // Id
    case 4:  // Course
    case 5:  // Group
      int i;
      if (!int.TryParse(s, out i))
        err = "Строку нельзя преобразовать в число";
```

```
      else
        if (i < 0)
          err = "Отрицательные числа не допускаются";
      break;
    case 1: // Surname
      if (s == "")
        err = "Поле \"Фамилия\" должно быть непустым";
      break;
    case 2: // Gender
      s = s.ToUpper();
      if (s != "М" && s != "Ж")
        err = "Допускаются значения \"М\" или \"Ж\"";
      break;
    case 3: // BirthDate
      if (s == "") break;
      DateTime d;
      if (!DateTime.TryParse(s, out d))
        err = "Строку нельзя преобразовать в дату";
      else
      {
        int age = DateTime.Today.Year - d.Year;
        if (age < 15 || age > 35)
          err = "Текущий возраст студента " +
            "должен находиться в диапазоне 15-35 лет";
      }
      break;
  }
  e.Cancel = err != "";
  dataGridView1.Rows[e.RowIndex].ErrorText = err;
}
```

Результат. Теперь при попытке выхода из измененного поля выполняется проверка введенного в него значения, и если это значение не является допустимым, то в заголовке соответствующей строки выводится значок ошибки  , при наведении на который появляется всплывающая подсказка, объясняющая причину ошибки. При наличии символа ошибки нельзя выйти из режима редактирования и, следовательно, перейти на другую ячейку таблицы (однако

можно нажать клавишу <Esc>, чтобы вернуться к предыдущему, правильному значению текущего поля).

Для полей **Id**, **Course**, **Group** проверяется, что введенный в них текст можно преобразовать в целое число, причем это число должно быть неотрицательным; поле **Surname** должно быть непустым; поле **Gender** должно содержать либо строку `"М"`, либо строку `"Ж"` (в любом регистре); поле **BirthDate** должно либо быть пустым, либо содержать верную дату рождения, причем в последнем случае текущий возраст, определяемый этой датой, должен лежать в диапазоне от 15 до 35 лет. См. также комментарии 1 и 2.

Ошибка. Выход из поля **Surname** *последней* строки (строки NewRow) также блокируется, что является неправильным, так как содержимое строки NewRow не соответствует никакому элементу данных.

Исправление. В начало метода `dataGridView1_CellValidating` (см. листинг 31.5) вставьте следующий фрагмент:

```
if (dataGridView1.Rows[e.RowIndex].IsNewRow)
  return;
```

Результат. Теперь проверка ячеек для последней строки таблицы не производится.

Недочет. По умолчанию поле **Surname** является пустым (точнее, имеет значение `null`, как любой объект ссылочного типа) и, следовательно, остается таким после добавления к таблице новой строки (например, в результате ввода некоторого числа в ячейку **Id** строки NewRow). Поэтому если в добавленной строке не переходить в ячейку, соответствующую фамилии, то ошибка, связанная с пустой фамилией, выявлена не будет. Отмеченный недочет можно было бы исправить, предусмотрев для свойства `Surname` класса `Student` непустое значение по умолчанию (например, `"<Фамилия И.О.>"` или `"?"`), как это сделано для свойства `Gender`. Другим вариантом исправления указанного недочета является проверка элемента данных, т. е. строки таблицы, *в целом*. Этот вариант будет реализован в следующем разделе.

Комментарии

1. Метод `dataGridView1_CellValidating` связан с событием `CellValidating`, которое возникает при выходе из текущей ячейки таблицы (при этом находиться в режиме редактирования необязательно). Вся необходимая информация о текущей ячейке передается через свойства второго параметра `e`: это индекс строки `RowIndex`, индекс столбца `ColumnIndex` и текущее значение ячейки `ValueFormatted`. Последнее свойство имеет тип `object`; его реаль-

ный тип зависит от типа ячейки (`string` для поля ввода, `bool` для ячейки-флажка). Для запрета выхода из текущей ячейки достаточно положить свойство `e.Cancel` равным `true`. Одновременно с этим желательно поместить в свойство `ErrorText` текущей строки текст с кратким описанием ошибки (непустое свойство `ErrorText` обеспечивает вывод значка ошибки в заголовке соответствующей строки).

2. Для проверки возможности преобразования содержимого строки к типу `int` или `DateTime` используется метод `TryParse` (ранее мы применяли этот метод в *разд. 6.2* для преобразования строк в вещественные числа). Если преобразование невозможно, то в варианте для типа `int` и других числовых типов второй (выходной) параметр будет содержать нулевое число, а в варианте для типа `DateTime` — значение `DateTime.MinValue`. Заметим, что мы воспользовались простейшим из двух вариантов метода `TryParse`, предусмотренных для числовых типов и типа `DateTime`.

31.3. Проверка правильности данных на уровне строки таблицы

Определите обработчик события `RowValidating` для таблицы `dataGridView1` (листинг 31.6).

Листинг 31.6. Обработчик `dataGridView1.RowValidating`

```
private void dataGridView1_RowValidating(object sender,
  DataGridViewCellCancelEventArgs e)
{
  if (dataGridView1.Rows[e.RowIndex].IsNewRow)
    return;
  string err = "";
  if (dataGridView1[1, e.RowIndex].Value == null)
    err = "Поле \"Фамилия\" должно быть непустым";
  e.Cancel = err != "";
  dataGridView1.Rows[e.RowIndex].ErrorText = err;
}
```

Результат. Теперь при наличии пустого поля **Surname** нельзя покинуть текущую строку; если же предпринимается такая попытка, то в заголовке строки отображается значок ошибки.

Комментарий

Контроль данных на уровне строки позволяет отказаться от блокирования выхода из ошибочной *ячейки* строки. Это дает возможность пользователю заполнять другие поля текущего элемента данных даже в случае, если некоторые из этих полей содержат неверные данные (необходимо лишь впоследствии вернуться к этим полям и откорректировать их значения; в противном случае покинуть текущую строку таблицы будет невозможно). При реализации подобного контроля за правильностью данных удобно выводить значок ошибки непосредственно в той ячейке, в которой ошибка обнаружена; для этого достаточно заносить сообщение об ошибке в свойство `ErrorText` не строки, а текущей *ячейки*. Отметим, что подобный вариант вывода значка имеет еще одно преимущество: значок ошибки исчезает сразу после начала редактирования ошибочной ячейки, в то время как значок в заголовке строки исчезнет только после ввода правильного значения и выхода из данной ячейки.

При обработке ошибок на уровне строки таблицы надо принимать во внимание три обстоятельства:

- все ячейки таблицы связаны с соответствующими свойствами класса, определяющего набор данных (в нашем примере — класса `Student`), поэтому в случае ошибок ввода, не позволяющих преобразовать введенный текст к типу соответствующего свойства, все же придется блокировать выход из текущей ячейки;
- при блокировании выхода из ячейки отобразить в ней значок ошибки невозможно, поэтому в случае ошибки, не позволяющей выйти из текущей ячейки, следует выводить значок ошибки в заголовке строки;
- в обработчике события `RowValidating` потребуется просматривать все ячейки текущей строки для поиска сообщений об ошибках. В листинге 31.7 приводится пример соответствующего кода.

Листинг 31.7. Фрагмент программы, предназначенный для поиска сообщений об ошибках для ячеек текущей строки таблицы

```
foreach (DataGridViewCell c in dataGridView1.Rows[e.RowIndex].Cells)
  if (c.ErrorText != "")
  {
    e.Cancel = true;
    return;
  }
```

31.4. Дополнительная настройка столбцов таблицы, использование выпадающих списков для текстовых полей

При организации ввода данных в таблицу важно не только предусмотреть средства проверки их правильности, но и обеспечить максимальную простоту ввода допустимых данных. Если набор вариантов для какого-либо поля невелик, то для выбора требуемого варианта удобно использовать *выпадающий список*. Настроим соответствующим образом столбец, связанный с полем `Gender`, изменив тип этого столбца на `DataGridViewComboBoxColumn`. Кроме того, откорректируем имена, заголовки и относительную ширину столбцов таблицы.

Перейдите в окне **Properties** для таблицы `dataGridView1` на свойство `Columns` и нажмите кнопку с многоточием [...] рядом с этим свойством. В результате на экране появится диалоговое окно **Edit Columns** настройки свойств столбцов таблицы. Последовательно выделяя имя каждого столбца в левом списке, откорректируйте требуемые свойства выделенного столбца (листинг 31.8). Дополнительно определите свойство `Items` столбца `gender1`. Для определения этого свойства используется специальное диалоговое окно (см. рис. 17.2), в которое надо ввести символы м и ж, набирая каждый символ на отдельной строке:

```
М
Ж
```

Завершив определение свойств, закройте окно **Edit Columns**, нажав клавишу <Enter> или кнопку **OK**. В результате форма `Form1` примет вид, приведенный на рис. 31.3.

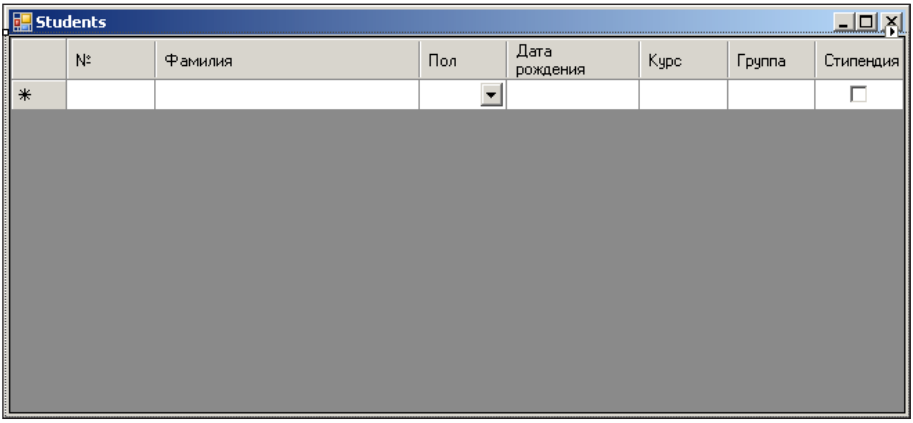


Рис. 31.3. Вид формы `Form1` для проекта STUD1 на промежуточном этапе разработки

В дальнейшем при настройке свойств столбцов мы будем ссылаться на их имена, определяемые свойством Name (например, столбец id1, столбец surname1 и т. д.).

Листинг 31.8. Настройка свойств столбцов таблицы dataGridView1

```
Id: Name = id1, HeaderText = №
Surname: Name = surname1, HeaderText = Фамилия,
  FillWeight = 300
Gender: Name = gender1, HeaderText = Пол,
  ColumnType = DataGridViewComboBoxColumn
BirthDate: Name = birthDate1, HeaderText = Дата рождения,
  FillWeight = 150
Course: Name = course1, HeaderText = Курс
Group: Name = group1, HeaderText = Группа
Scholarship: Name = scholarship1, HeaderText = Стипендия
```

Результат. Теперь заголовки столбцов таблицы содержат русский текст. Ширина столбца **Фамилия** увеличена в 3 раза по сравнению с шириной прочих столбцов, а ширина столбца **Дата рождения** — в 1,5 раза. Столбец **Пол** позволяет выбирать значения только из выпадающего списка. Отметим, что ввести в столбец **Пол** значение с помощью клавиатуры можно по крайней мере тремя способами: либо набрать требуемую букву ("м" или "ж", регистр не важен), либо развернуть выпадающий список с помощью клавиатурной комбинации <Alt>+<↓> и выбрать требуемый вариант с помощью клавиш <↓> и <↑>, после чего нажать клавишу <Enter>, либо, наконец, перейти в режим редактирования, нажав клавишу <F2> и, *не разворачивая список*, выбрать нужный вариант с помощью клавиш <↓> и <↑>.

Комментарии

1. Имена столбцов таблицы, определяемые их свойством Name, удобно использовать для доступа к этим столбцам, поскольку коллекция Columns допускает индексирование не только по номеру столбца, но и по его имени, например, Columns["surname1"]. Заметим, что при автоматической генерации столбцов таблицы в режиме дизайна им (как и пунктам меню — *см. разд. 18.1*) присваиваются чрезмерно длинные имена, например, surnameDataGridViewTextBoxColumn.
2. Для настройки относительной ширины столбцов предназначено свойство FillWeight (вещественного типа float). По умолчанию его значение

для всех столбцов равно **100**. Свойство FillWeight влияет на ширину столбца, если столбец имеет *режим выравнивания* Fill. Режим выравнивания можно настроить для отдельного столбца с помощью его свойства AutoSizeMode, а также для всех столбцов одновременно с помощью свойства AutoSizeColumnsMode компонента DataGridView, как сделано в нашем примере (см. листинг 31.1).

3. Выпадающие списки удобно использовать и для *числовых* полей, если диапазон их допустимых значений невелик (например, для поля **Курс** диапазон может включать числа от 1 до 6, а также число 0, означающее, что курс не задан). Однако задавать требуемые элементы выпадающего списка с помощью свойства-коллекции Items в режиме дизайна нельзя, так как при определении коллекции Items в этом режиме предполагается, что все ее элементы имеют тип string, а этот тип не может быть автоматически преобразован к типу числового поля. Специальный механизм, позволяющий использовать выпадающие списки для нетекстовых полей, будет рассмотрен в следующей главе *(см. разд. 32.3)*.

31.5. Использование XML-сериализации для сохранения и загрузки наборов данных

Добавьте в форму Form1 невизуальные компоненты типа OpenFileDialog и SaveFileDialog, а также компонент-меню MenuStrip (эти компоненты получат имена openFileDialog1, saveFileDialog1 и menuStrip1; они будут размещены в области невизуальных компонентов под изображением формы, кроме того, в верхней части формы появится строка меню, связанная с компонентом menuStrip1).

Используя конструктор меню *(см. разд. 18.1)*, создайте в компоненте menuStrip1 пункт меню первого уровня с текстом **&File** и с помощью окна **Properties** измените имя этого пункта (т. е. свойство Name) на **file1**. В выпадающем меню, связанном с пунктом **File**, создайте три пункта меню с текстом **&New**, **&Open...** и **Save &As...**. Настройте свойства добавленных компонентов и пунктов меню (листинг 31.9; рис. 31.4).

В начало файла Form1.cs добавьте следующие операторы:

```
using System.IO;
using System.Xml.Serialization;
```

В описание класса Form1 добавьте новое поле:

```
private XmlSerializer xmls = new XmlSerializer(typeof(List<Student>));
```

Дополните конструктор класса Form1 (листинг 31.10).

Опишите в классе `Form1` новый метод `SaveData` (листинг 31.11) и определите обработчики события `DropDownOpening` для пункта меню `file1`, события `Click` для пунктов меню `new1`, `open1` и `saveAs1`, а также события `FormClosing` для формы `Form1` (листинг 31.12).

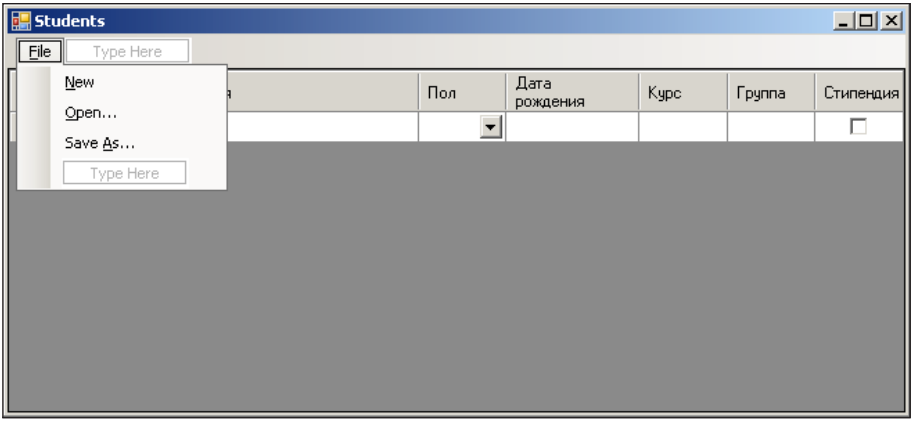


Рис. 31.4. Вид формы `Form1` для проекта STUD1 после добавления строки меню

Листинг 31.9. Настройка свойств

```
openFileDialog1, saveFileDialog1: DefaultExt = studXml,
  Filter = Student data files|*.studXml
пункт меню New (группа File): Name = new1
пункт меню Open (группа File): Name = open1
пункт меню Save As... (группа File): Name = saveAs1
```

Листинг 31.10. Новый вариант конструктора формы `Form1`

```
public Form1()
{
  InitializeComponent();
  bindingSource1.DataSource = new List<Student>();
}
```

Листинг 31.11. Метод `SaveData` формы `Form1`

```
private void SaveData(string name)
{
  if (name == "" || dataGridView1.RowCount == 1)
```

```
    return;
  StreamWriter sw = new StreamWriter(name, false, Encoding.Default);
  xmls.Serialize(sw, bindingSource1.DataSource);
  sw.Close();
}
```

Листинг 31.12. Обработчики `file1.DropDownOpening`, `new1.Click`, `open1.Click`, `saveAs1.Click`, `Form1.FormClosing`

```
private void file1_DropDownOpening(object sender, EventArgs e)
{
  saveAs1.Enabled = dataGridView1.RowCount > 1;
}
private void new1_Click(object sender, EventArgs e)
{
  SaveData(saveFileDialog1.FileName);
  bindingSource1.DataSource = new List<Student>();
  dataGridView1.CurrentCell = dataGridView1[0, 0];
  saveFileDialog1.FileName = "";
  Text = "Students";
}
private void open1_Click(object sender, EventArgs e)
{
  openFileDialog1.FileName = "";
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    SaveData(saveFileDialog1.FileName);
    string s = openFileDialog1.FileName;
    StreamReader sr = new StreamReader(s, Encoding.Default);
    bindingSource1.DataSource = xmls.Deserialize(sr);
    sr.Close();
    saveFileDialog1.FileName = s;
    Text = "Students - " + Path.GetFileNameWithoutExtension(s);
  }
}
private void saveAs1_Click(object sender, EventArgs e)
{
```

```
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
  {
    string s = saveFileDialog1.FileName;
    SaveData(s);
    Text = "Students - " + Path.GetFileNameWithoutExtension(s);
  }
}
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
  SaveData(saveFileDialog1.FileName);
}
```

Результат. Добавленные команды меню **File** позволяют сохранить текущий непустой набор данных в файле с указанным именем (команда **Save As**), прочесть из файла ранее сохраненный набор данных (команда **Open**) и создать пустой набор данных (команда **New**). Имя файла, связанного с текущим набором данных, отображается в заголовке окна после текста **Students** (если с набором данных еще не связан файл, заголовок окна содержит только текст **Students**). При выполнении команд **New** и **Open**, а также при выходе из программы предыдущий набор данных автоматически сохраняется (при условии, что этот набор является непустым и с ним уже связан файл). При создании пустого набора данных текущей становится первая ячейка первой строки.

> ***ПРИМЕЧАНИЕ***
>
> Имя файла, связанного с текущим набором данных, хранится в свойстве `FileName` компонента `saveFileDialog1`; если это свойство является пустой строкой, значит, текущий набор данных еще не связан с файлом.

Файлы, содержащие наборы данных, имеют специальный текстовый формат, называемый *XML-форматом*; чтобы это подчеркнуть, для них выбрано нестандартное расширение studXml. В качестве примера приведем содержимое файла с набором данных, изображенным на рис. 31.2 (листинг 31.13). См. также комментарий 1.

Листинг 31.13. Пример XML-файла с набором данных

```
<?xml version="1.0" encoding="windows-1251"?>
<ArrayOfStudent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 ↳ xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Student>
```

```
    <Id>1</Id>
    <Surname>Иванов И.И.</Surname>
    <Gender>M</Gender>
    <BirthDate>1991-03-12T00:00:00</BirthDate>
    <Course>1</Course>
    <Group>2</Group>
    <Scholarship>true</Scholarship>
  </Student>
  <Student>
    <Id>2</Id>
    <Surname>Петров П.П.</Surname>
    <Gender>M</Gender>
    <BirthDate>1989-01-23T00:00:00</BirthDate>
    <Course>3</Course>
    <Group>1</Group>
    <Scholarship>false</Scholarship>
  </Student>
  <Student>
    <Id>3</Id>
    <Surname>Сергеева С.С.</Surname>
    <Gender>Ж</Gender>
    <BirthDate>1988-05-05T00:00:00</BirthDate>
    <Course>4</Course>
    <Group>3</Group>
    <Scholarship>true</Scholarship>
  </Student>
</ArrayOfStudent>
```

ПРИМЕЧАНИЕ

В коде используется значок переноса строки ↳, который указывает, что эта строка является продолжением предыдущей, и разрыва строки здесь быть не должно. Разумеется, вводить этот значок при наборе кода не надо.

Ошибка 1. Если текущей строкой является последняя строка таблицы (строка NewRow), то попытка прочесть набор данных из файла приводит к ошибке времени выполнения. Эта ошибка объясняется тем, что перед загрузкой нового набора данных между таблицей и компонентом `bindingSource1` проис-

ходит попытка обмена данных, взятых из текущей строки, которая завершается неудачей, поскольку строка NewRow не связана с реальным элементом набора данных.

Исправление. Добавьте в метод `open1_Click` два новых оператора (листинг 31.14).

Листинг 31.14. Новый вариант метода `open1_Click`

```
private void open1_Click(object sender, EventArgs e)
{
  openFileDialog1.FileName = "";
  if (openFileDialog1.ShowDialog() == DialogResult.OK)
  {
    SaveData(saveFileDialog1.FileName);
    string s = openFileDialog1.FileName;
    StreamReader sr = new StreamReader(s, Encoding.Default);
    bindingSource1.SuspendBinding();
    bindingSource1.DataSource = xmls.Deserialize(sr);
    bindingSource1.ResumeBinding();
    sr.Close();
    saveFileDialog1.FileName = s;
    Text = "Students - " + Path.GetFileNameWithoutExtension(s);
  }
}
```

Результат. Перед загрузкой нового набора данных компонент `bindingSource1` временно отсоединяется от компонента `dataGridView1`, поэтому никакие действия, связанные с обработкой текущей ячейки, в ходе загрузки не выполняются. После загрузки соединение восстанавливается.

Ошибка 2. Если при записи набора данных текущей ячейкой является одна из ячеек последней строки (строки NewRow), то в файл, кроме "настоящих" элементов набора данных, записывается дополнительный элемент с данными, взятыми из последней строки таблицы (этот элемент, в частности, будет содержать пустое поле **Фамилия**). Как и в случае предыдущей ошибки, причина заключается в особой обработке текущей строки, которая в данной ситуации не связана с реальным элементом набора данных.

Исправление. Измените метод `SaveData` (листинг 31.15).

Листинг 31.15. Новый вариант метода SaveData

```
private void SaveData(string name)
{
  if (name == "" || dataGridView1.RowCount == 1)
    return;
  if (dataGridView1.CurrentRow.IsNewRow)
    dataGridView1.CurrentCell =
      dataGridView1[0, dataGridView1.RowCount - 2];
  StreamWriter sw = new StreamWriter(name, false, Encoding.Default);
  xmls.Serialize(sw, bindingSource1.DataSource);
  sw.Close();
}
```

Результат. Перед сохранением набора данных выполняется проверка, является ли текущая строка строкой NewRow, и если является, то текущей ячейкой делается первая ячейка предыдущей строки (заметим, что предыдущая строка всегда существует, так как ранее в методе `SaveData` проверяется, что таблица содержит хотя бы две строки).

Недочет. Если при выполнении любой из команд меню таблица находится в режиме редактирования, то сохраненный набор данных будет содержать "старые" данные для той ячейки, которая редактировалась в момент сохранения (поскольку передача нового значения в компонент `bindingSource1` происходит при выходе из режима редактирования).

Исправление. Определите обработчик события `CurrentCellDirtyStateChanged` для компонента `dataGridView1` (листинг 31.16).

Листинг 31.16. Обработчик dataGridView1.CurrentCellDirtyStateChanged

```
private void  dataGridView1_CurrentCellDirtyStateChanged(object sender,
  EventArgs e)
{
  menuStrip1.Enabled = !dataGridView1.IsCurrentCellDirty;
}
```

Результат. Теперь при переходе в режим редактирования и изменении значения текущей ячейки компонент `menuStrip1` становится недоступным, и, следовательно, блокируются любые команды меню. Доступность меню восстанавливается при выходе из режима редактирования. Заметим, что при закрытии

окна программы проблем, связанных с отмеченным недочетом, не возникает: даже если в момент закрытия окна какая-либо ячейка находилась в режиме редактирования, перед выполнением обработчика `FormClosing` (в котором выполняется сохранение набора данных) таблица автоматически *выходит* из режима редактирования; таким образом, измененные данные успешно сохраняются в файле. Если же из-за ошибок ввода выход из режима редактирования невозможен, то автоматически блокируются действия по закрытию окна. См. также комментарии 2 и 3.

Комментарии

1. Формат XML в настоящее время является наиболее распространенным текстовым форматом для хранения наборов данных, поэтому для работы с этим форматом в .NET Framework предусмотрен ряд классов. Одним из простейших в использовании является класс `XmlSerializer` *(XML-сериализатор)*, позволяющий сохранять и считывать из файла данные, связанные с отдельным объектом или коллекцией объектов. При создании объекта типа `XmlSerializer` необходимо указать тип того объекта, данные которого требуется записать в файл или прочесть из файла (данными служат все открытые поля, а также открытые свойства объекта, доступные одновременно для чтения и записи). Метод `Serialize` записывает данные из объекта в файл, а метод `Deserialize` создает объект указанного типа и заполняет его данными, прочитанными из файла. В программе использован вариант этих методов, у которого в качестве первого параметра указывается файловый поток (типа `StreamReader` или `StreamWriter`). Сериализуемым объектом является список `List<Student>` (т. е. список объектов `Student`), который связан с компонентом `bindingSource1` и хранится в его свойстве `DataSource` (см. листинг 31.10).
2. Свойство таблицы `IsCurrentCellDirty` равно `true`, когда значение текущей ячейки уже изменено, однако еще не зафиксировано. Фиксация изменений (committing) происходит при выходе из режима редактирования при условии, что эти изменения успешно прошли проверку на правильность.
3. Имеется вариант исправления отмеченного недочета, не требующий блокирования меню: достаточно перед сохранением набора данных вызывать метод `EndEdit` объекта `dataGridView1`, который обеспечивает немедленный выход из режима редактирования и пересылку новых данных в компонент `bindingSource1`. Правда, при реализации подобного варианта потребовалось бы особым образом обрабатывать ситуацию, когда нормальное завершение редактирования невозможно из-за ввода в ячейку ошибочных данных.

4. При исправлении отмеченного недочета более естественным представляется вариант, использующий обработчики событий `CellBeginEdit` и `CellEndEdit` компонента `dataGridView1` (в первом обработчике свойство `menuStrip1.Enabled` надо положить равным `false`, а во втором — `true`). Однако такой вариант будет работать не вполне правильно из-за странной реакции ячеек типа `DataGridViewCheckBoxCell` на событие `CellBeginEdit` (в нашей таблице такие ячейки расположены в столбце **Стипендия**). Для ячейки указанного типа событие `CellBeginEdit` наступает *при простом переходе на нее*, т. е. в тот момент, когда она делается текущей. Поэтому переход меню в неактивное состояние будет происходить при простом *выделении* одной из ячеек указанного типа, что вызовет недоумение у пользователя программы. Таким образом, подобное поведение ячеек типа `DataGridViewCheckBoxCell` вынуждает нас отказаться от использования обработчиков `CellBeginEdit` и `CellEndEdit`. Впрочем, можно вместо этого отказаться от использования ячеек типа `DataGridViewCheckBoxCell`, заменив их выпадающими списками из двух значений.

31.6. Дополнительные средства навигации и редактирования для таблицы с набором данных

Добавьте в форму `Form1` компонент типа `BindingNavigator` (он получит имя `bindingNavigator1` и будет размещен в области невизуальных компонентов под изображением формы, кроме того, в верхней части формы появится панель инструментов, связанная с этим компонентом). Для компонента `bindingNavigator1` установите свойство `Dock` равным **`Bottom`** (в результате панель переместится в нижнюю часть формы), а свойство `BindingSource` равным **`bindingSource1`**.

Для компонента `bindingNavigator1` необходимо также выполнить команду **Send to Back** (нажав кнопку  на панели **Layout** или воспользовавшись контекстным меню данного компонента). Это необходимо для того, чтобы панель инструментов не заслоняла нижнюю часть таблицы `dataGridView1`. В результате форма `Form1` примет вид, приведенный на рис. 31.5.

Результат. С помощью элементов управления, размещенных на панели `bindingNavigator1`, можно быстро перемещаться на первую (), предыдущую (), последующую (), последнюю () строку таблицы, а также на строку таблицы с указанным номером (). Кроме того, можно добавлять

к таблице новые строки ( ; новая строка добавляется в конец таблицы, непосредственно перед строкой NewRow) и удалять существующие ( ; удаляется строка, содержащая текущую ячейку).

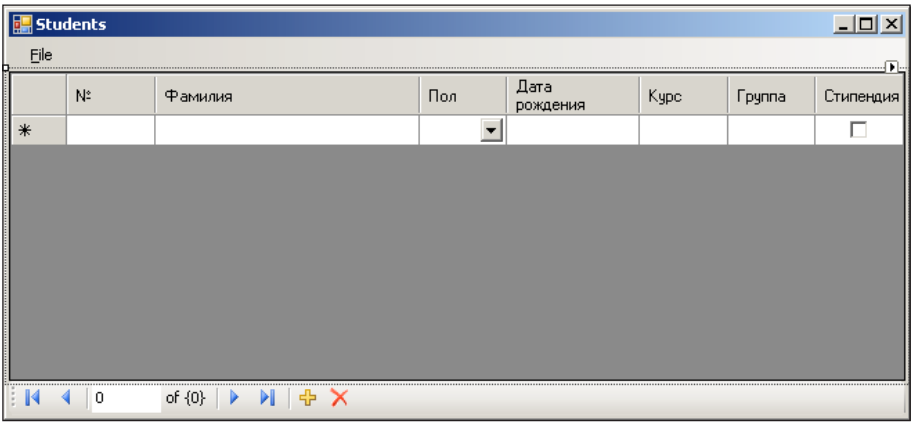


Рис. 31.5. Окончательный вид формы `Form1` для проекта STUD1

Недочет. Если текущей является строка NewRow, то действие по удалению выполняется не вполне естественным образом: удаляется строка, *предшествующая* строке NewRow. Кроме того, при попытке удалить единственную строку NewRow в *пустом* наборе данных возникает ошибка времени выполнения.

Исправление. Определите обработчик события `RowEnter` для таблицы `dataGridView1` (листинг 31.17).

Листинг 31.17. Обработчик `dataGridView1.RowEnter`

```
private void dataGridView1_RowEnter(object sender,
  DataGridViewCellEventArgs e)
{
  bindingNavigatorDeleteItem.Enabled =
    !dataGridView1.Rows[e.RowIndex].IsNewRow;
}
```

Результат. Теперь в случае, когда текущей является последняя строка таблицы, кнопка удаления недоступна.

ПРИМЕЧАНИЕ

Читатель, вероятно, уже заметил, как много дополнительных проблем возникает из-за наличия в таблице особой строки NewRow. Для того чтобы не сталкиваться

с подобными проблемами, можно просто отказаться от использования этой строки, положив свойство AllowUserToAddRows таблицы dataGridView1 равным **False**. Такой шаг не приведет к потере функциональности, так как на панели bindingNavigator1 предусмотрена кнопка для добавления в набор данных нового элемента. Тем не менее, наличие строки NewRow дает возможность добавлять новые элементы более быстрым и наглядным способом, поэтому мы сохраним ее в нашем проекте и его модификации, описанной в *главе 32*.

Комментарий

Компонент BindingNavigator является специализированной панелью инструментов, на которую уже добавлены стандартные кнопки для управления набором данных. Вполне допустимо дополнять эту панель новыми кнопками и даже удалять из нее те стандартные кнопки, которые не требуются для разрабатываемого приложения. Кроме того, можно изменять поведение стандартных кнопок. Например, вместо приведенного варианта исправления отмеченного недочета (см. листинг 31.17) можно было бы изменить поведение кнопки удаления так, чтобы при попытке удалить последнюю строку таблицы она не выполняла никаких действий. Для этого достаточно явным образом определить обработчик события Click для кнопки bindingNavigatorDeleteItem, а также *очистить* свойство DeleteItem компонента bindingNavigator1 (чтобы отключить от кнопки bindingNavigatorDeleteItem стандартные действия по удалению, предусмотренные в компоненте bindingNavigator1).

31.7. Автоматизация действий при добавлении нового элемента в набор данных

Настройте свойства первого столбца таблицы dataGridView1 (листинг 31.18). Напомним, что для доступа к свойствам столбца (в данном случае id1) в режиме дизайна достаточно щелкнуть на кнопке с многоточием  рядом со свойством Columns компонента dataGridView1. Свойство DefaultCellStyle столбца включает много "подсвойств", для их отображения надо щелкнуть на кнопке с многоточием рядом с этим свойством. Отметим, наконец, что задать указанный цвет **GrayText** можно, либо введя его имя, используя клавиатуру, либо развернув список цветов и выбрав этот цвет на вкладке **System**.

Дополните метод dataGridView1_CurrentCellDirtyStateChanged (листинг 31.19).

Листинг 31.18. Настройка свойств

```
dataGridView1.Columns.id1: ReadOnly = True,
  DefaultCellStyle.ForeColor = GrayText,
  DefaultCellStyle.SelectionForeColor = GrayText
```

Листинг 31.19. Новый вариант метода dataGridView1_CurrentCellDirtyStateChanged

```
private void  dataGridView1_CurrentCellDirtyStateChanged(object sender,
  EventArgs e)
{
  menuStrip1.Enabled = !dataGridView1.IsCurrentCellDirty;
  int row = dataGridView1.CurrentRow.Index;
  if (!dataGridView1.IsCurrentCellDirty &&
    (int)dataGridView1["Id1", row].Value == 0)
  {
    int maxId = 0;
    for (int i = 0; i < row; i++)
    {
      int v = (int)dataGridView1["Id1", i].Value;
      if (maxId < v)
        maxId = v;
    }
    dataGridView1["Id1", row].Value = maxId + 1;
  }
}
```

Результат. При добавлении нового элемента данных (точнее, после завершения редактирования одного из его полей) в его свойство `Id`, отображаемое в первом столбце таблицы с заголовком **№**, автоматически заносится номер, который на 1 больше максимального из уже имеющихся номеров в загруженном наборе данных. Поскольку теперь столбец `id1` заполняется автоматически, в его явном редактировании нет необходимости, и поэтому столбец `id1` сделан столбцом только для чтения (см. листинг 31.18). Чтобы визуально подчеркнуть это свойство столбца `id1`, его текст как в выделенном, так и в обычном состоянии отображается серым цветом.

Приведем изображение работающей программы (рис. 31.6). Таблица находится в режиме редактирования, поэтому меню недоступно.

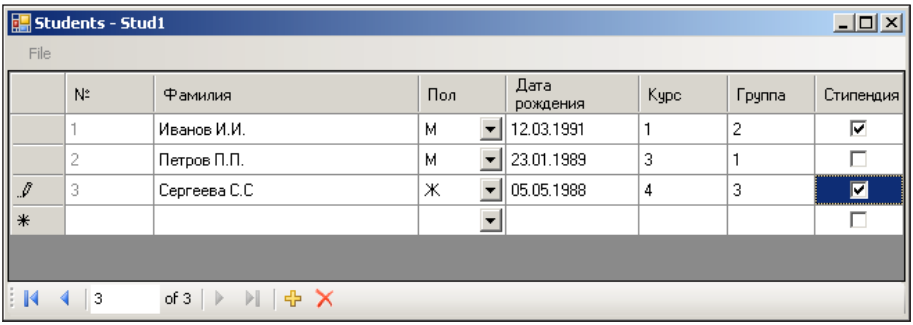


Рис. 31.6. Вид работающего приложения STUD1

Комментарии

1. Реализованное в настоящем разделе поведение столбца характерно для полей-*счетчиков* таблиц баз данных: эти поля тоже определяются автоматически и не допускают явного редактирования. Основное назначение полей-счетчиков — обеспечить уникальность каждой записи в таблице. Кроме того, с их помощью можно определить, в каком порядке данные заносились в таблицу.
2. Так как редактировать данные в первом столбце таблицы нельзя, было бы естественно запретить переход на ячейки этого столбца. К сожалению, подобная возможность в компоненте `DataGridView` не предусмотрена. Можно лишь изменить внешний вид ячеек столбца. Мы выводим текст этих ячеек системным цветом `GrayText`, который традиционно используется при отображении элементов, недоступных для редактирования.

Глава 32

Дополнительные возможности, связанные с обработкой наборов данных: проект STUD2

Проект STUD2 посвящен дополнительным средствам, связанным с обработкой наборов данных и их привязкой к компонентам формы. В нем демонстрируется создание *подчиненных наборов данных*, связываемых с элементами основного набора и отображаемых в дополнительных (подчиненных) таблицах; описываются средства, позволяющие связывать элементы набора данных с произвольными визуальными компонентами (не обязательно таблицами); рассматриваются дополнительные возможности, связанные с таблицей `DataGridView` (использование выпадающих списков для отображения полей с нетекстовой информацией, использование столбцов с вычисляемыми полями). Кроме того, рассматривается компонент `TabControl` и демонстрируются способы сортировки и поиска данных, основанные на применении функций сравнения (comparison) и тестовых функций (predicate).

32.1. Использование набора вкладок и добавление таблицы с подчиненными данными

В качестве заготовки для проекта STUD2 следует использовать ранее разработанный проект STUD1 *(см. главу 31)*. Скопируйте проект STUD1 в новый каталог STUD2 и выполните действия, необходимые для переименования проекта *(см. разд. 1.1)*. Разместите в форме `Form1` компонент типа `TabControl` (он получит имя `tabControl1`). Напомним, что для добавления нового компонента в форму (даже если она целиком заполнена другими компонентами) достаточно после выбора компонента в окне **Toolbox** щелкнуть мышью на *заголовке* формы.

После добавления компонента `tabControl1` он уже будет содержать две вкладки с именами `tabPage1` и `tabPage2` (см. комментарий 1).

Загрузите в редактор файл Form1.Designer.cs, найдите в нем оператор

```
this.Controls.Add(this.dataGridView1);
```

и измените его следующим образом:

```
this.tabPage1.Controls.Add(this.dataGridView1);
```

Это изменение обеспечит отображение компонента `dataGridView1` не в самой форме, а в компоненте-контейнере `tabPage1` (см. комментарий 2).

Настройте свойства компонента `tabControl1` и его дочерних компонентов `tabPage1` и `tabPage2` (см. первые три строки в листинге 32.1, а также рис. 31.1). Заметим, что для выделения компонента `tabPage1` достаточно выделить таблицу `dataGridView1`, целиком заполняющую этот компонент, после чего нажать клавишу <Esc>.

Создайте новый файл Mark.cs с описанием класса `Mark` (листинг 32.2; действия, связанные с созданием нового класса и определением его полей и свойств, описаны в *разд. 31.1*).

Перейдите в файл Student.cs и в описание класса `Student` добавьте новое свойство `Marks` вместе со связанным с ним полем (листинг 32.3). Свойство `Marks` имеет тип `List<Mark>` и позволяет хранить набор оценок студента по различным предметам. Поскольку это свойство, в отличие от прочих свойств, нельзя представить в виде одного столбца таблицы, свяжем его с дополнительной *(подчиненной)* таблицей, в каждой строке которой будем выводить информацию об одной из оценок, полученных текущим студентом.

Добавьте в форму `Form1` еще один компонент типа `BindingSource` (он получит имя `bindingSource2`), разместите на вкладке `tabPage2` компонента `tabControl1` еще одну таблицу (с именем `dataGridView2`) и настройте свойства добавленной таблицы (листинг 32.1, строка `dataGridView2`).

Поскольку таблица `dataGridView2` должна быть связана с *частью* данных класса `Student`, свойство `DataSource` компонента `bindingSource2` нельзя задать в режиме дизайна (чуть позже мы определим это свойство в конструкторе класса `Form1` — см. листинг 32.4). Поэтому нам потребуется явным образом добавить к таблице `dataGridView2` столбцы и настроить все их свойства.

Выполняя действия, аналогичные тем, которые были описаны в *разд. 30.1*, добавьте в таблицу `dataGridView2` два столбца с именами **`subject1`** и **`level1`** и настройте их свойства (листинг 32.1; рис. 32.2). Заметим, что значения свойства `DataPropertyName` необходимо ввести, используя клавиатуру, так как связанный с этим свойством выпадающий список нужных значений не содержит.

В конструктор класса `Form1` добавьте новые операторы (листинг 32.4).

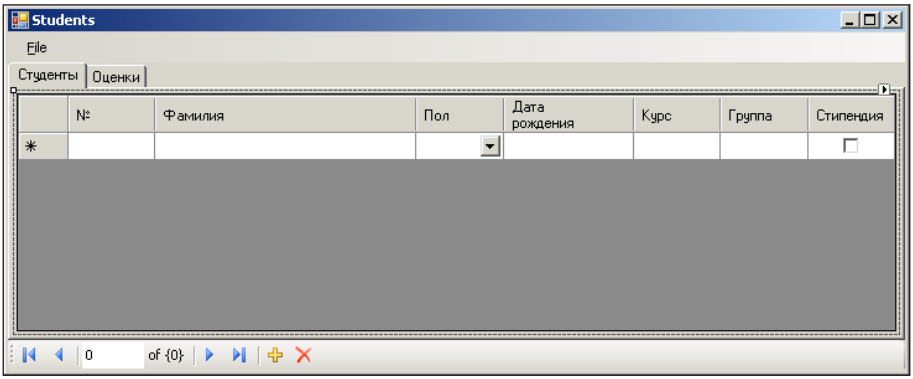


Рис. 32.1. Вид формы `Form1` для проекта STUD2 на начальном этапе разработки (вкладка **Студенты**)

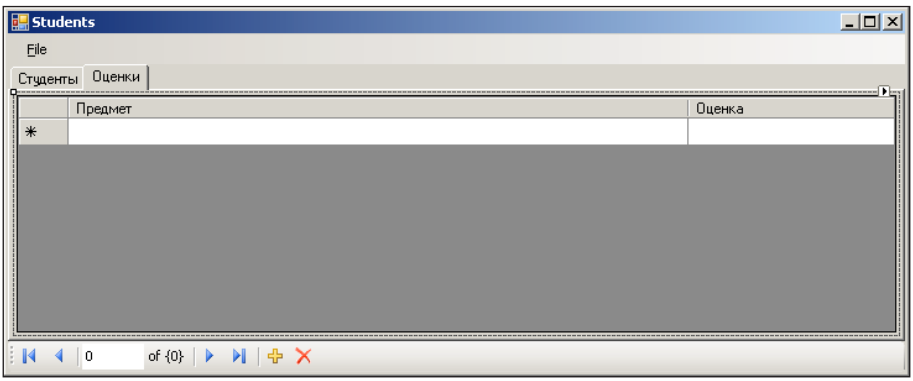


Рис. 32.2. Вид формы `Form1` для проекта STUD2 на начальном этапе разработки (вкладка **Оценки**)

Листинг 32.1. Настройка свойств компонентов, а также столбцов таблицы `dataGridView2`

```
tabControl1: Dock = Fill
tabPage1: Text = Студенты
tabPage2: Text = Оценки
dataGridView2: Dock = Fill, AutoSizeColumnsMode = Fill,
  DataSource = bindingSource2
subject1: HeaderText = Предмет, DataPropertyName = Subject,
  FillWeight = 300
level1: HeaderText = Оценка, DataPropertyName = Level
```

Листинг 32.2. Текст файла Mark.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace STUD1
{
  public class Mark
  {
    private string subject;
    public string Subject
    {
      get { return subject; }
      set { subject = value; }
    }
    private int level;
    public int Level
    {
      get { return level; }
      set { level = value; }
    }
  }
}
```

Листинг 32.3. Добавление к описанию класса Student из файла Student.cs

```
private List<Mark> marks;
public List<Mark> Marks
{
  get { return marks; }
  set { marks = value; }
}
```

Листинг 32.4. Новый вариант конструктора формы Form1

```
public Form1()
{
  InitializeComponent();
```

```
  bindingSource1.DataSource = new List<Student>();
  bindingSource2.DataMember = "Marks";
  bindingSource2.DataSource = bindingSource1;
}
```

Результат. Мы можем вводить в подчиненную таблицу (размещенную на вкладке **Оценки**) названия предметов и оценки по ним для текущего студента. При сохранении созданного набора данных в XML-файле в этот файл записываются сведения не только из основной таблицы **Студенты**, но и из всех подчиненных таблиц **Оценки** (подчеркнем, что с каждым студентом связывается *своя* подчиненная таблица с оценками). Например, если для студента Иванова И. И. в подчиненную таблицу были занесены две оценки, то в XML-файле данные о нем будут иметь вид, приведенный в листинге 32.5.

С помощью компонента `bindingNavigator1`, расположенного в нижней части формы, можно перемещаться по основному набору данных, находясь в *любой* из вкладок компонента `tabControl1`.

> ***ПРИМЕЧАНИЕ***
>
> Обратите внимание на то, что компонент `bindingSource2` связывается с частью данных из компонента `bindingSource1` (а именно со свойством-списком `Marks` объекта `Student`). Поэтому при изменении данных, подключаемых к `bindingSource1` (например, при выполнении команды **New** или **Open**), автоматически происходит и обновление данных для компонента `bindingSource2`.

Листинг 32.5. Фрагмент XML-файла с набором данных

```
<Student>
  <Id>1</Id>
  <Surname>Иванов И.И.</Surname>
  <Gender>M</Gender>
  <BirthDate>1991-03-12T00:00:00</BirthDate>
  <Course>1</Course>
  <Group>2</Group>
  <Scholarship>true</Scholarship>
  <Marks>
    <Mark>
      <Subject>Мат. анализ</Subject>
      <Level>5</Level>
```

```
    </Mark>
    <Mark>
      <Subject>Алгебра и геометрия</Subject>
      <Level>4</Level>
    </Mark>
  </Marks>
</Student>
```

Недочет 1. Подчиненную таблицу можно заполнять и в ситуации, когда текущей строкой таблицы **Студенты** является последняя строка, не связанная с данными (строка NewRow).

Исправление. Дополните метод `dataGridView1_RowEnter` (листинг 32.6).

Листинг 32.6. Новый вариант метода `dataGridView1_RowEnter`

```
private void dataGridView1_RowEnter(object sender,
  DataGridViewCellEventArgs e)
{
  bindingNavigatorDeleteItem.Enabled =
    !dataGridView1.Rows[e.RowIndex].IsNewRow;
  if (bindingNavigatorDeleteItem.Enabled &&
    !tabControl1.TabPages.Contains(tabPage2))
    tabControl1.TabPages.Add(tabPage2);
  else
    if (!bindingNavigatorDeleteItem.Enabled &&
      tabControl1.TabPages.Contains(tabPage2))
      tabControl1.TabPages.Remove(tabPage2);
}
```

Результат. Теперь при переходе на последнюю строку таблицы **Студенты** ярлычок вкладки **Оценки** скрывается (что делает невозможным переход на эту вкладку), а при переходе на любую другую строку таблицы **Студенты** ярлычок вновь появляется. См. также комментарий 3.

Недочет 2. При запуске программы ярлычок вкладки **Оценки** остается видимым.

Исправление. Определите обработчик события `Load` для формы `Form1` (листинг 32.7).

Листинг 32.7. Обработчик `Form1.Load`

```
private void Form1_Load(object sender, EventArgs e)
{
  ActiveControl = dataGridView1;
}
```

Результат. Теперь при запуске программы таблица `dataGridView1` получает фокус ввода, что обеспечивает скрытие ярлычка для вкладки **Оценки**. См. также комментарий 4.

> ***ПРИМЕЧАНИЕ***
>
> Следует отметить, что попытка изменить свойство `ActiveControl` непосредственно в *конструкторе* класса `Form1` не приведет к желаемому результату.

Недочет 3. При добавлении в таблицу `dataGridView1` новой строки с данными о студенте ярлычок вкладки **Оценки** появится только после выхода из этой строки (поскольку действия, связанные с отображением/скрытием ярлычка **Оценки**, пока связаны только с событием `RowEnter` — см. листинг 32.6).

Исправление. В метод `dataGridView1_CurrentCellDirtyStateChanged`, последний вариант которого приведен в листинге 31.19, добавьте новый фрагмент (листинг 32.8).

Листинг 32.8. Добавление к методу `dataGridView1_CurrentCellDirtyStateChanged`

```
if (!tabControl1.TabPages.Contains(tabPage2) &&
  !dataGridView1.IsCurrentCellDirty)
  tabControl1.TabPages.Add(tabPage2);
```

Результат. Теперь ярлычок вкладки **Оценки** появляется сразу после заполнения одного из полей в строке с данными о новом студенте.

> ***ПРИМЕЧАНИЕ***
>
> Напомним, что использовать в подобной ситуации "более естественный" обработчик события `CellEndEdit` не следует из-за не вполне корректной реакции на это событие ячеек типа `DataGridViewCheckBoxCell` (см. комментарий 4 в *разд. 31.5*).

Недочет 4. При редактировании ячеек таблицы **Оценки** меню приложения не переходит в неактивное состояние.

Исправление. Определите обработчик события `CurrentCellDirtyStateChanged` для таблицы `dataGridView2` (листинг 32.9).

Листинг 32.9. Обработчик `dataGridView2.CurrentCellDirtyStateChanged`

```
private void dataGridView2_CurrentCellDirtyStateChanged(object sender,
  EventArgs e)
{
  menuStrip1.Enabled = !dataGridView2.IsCurrentCellDirty;
}
```

Комментарии

1. Для добавления и удаления вкладок в компоненте `TabControl` в режиме дизайна можно использовать команды **Add Tab** и **Remove Tab** контекстного меню этого компонента. Если щелкнуть мышью на внутренней области или границе какой-либо вкладки, то будет выделен компонент типа `TabPage`, "отвечающий" за выделенную вкладку. В этом случае для удаления выделенной вкладки достаточно нажать клавишу <Del>.
2. Если для компонента определены какие-либо обработчики событий, то для его переноса из одного компонента-контейнера в другой *не следует* пользоваться буфером обмена (в частности, клавиатурными комбинациями <Ctrl>+<X> и <Ctrl>+<V>), так как в результате вырезания и последующей вставки компонента у него будут потеряны все настройки, связанные с обработчиками событий; иными словами, в разделе **Events** окна свойств для этого компонента *все строки станут пустыми* (настройки раздела **Properties** сохранятся).

   Заметим, что если компонент имеет небольшой размер и не пристыкован к границам своего родительского компонента, то для его переноса в другой компонент-контейнер можно использовать обычное перетаскивание мышью.
3. Несмотря на наличие у класса `TabPage` метода `Hide`, этот метод не позволяет скрыть ярлычок соответствующей вкладки (похоже, что метод `Hide` просто ничего не делает). В документации к компоненту `TabControl` отмечается, что единственный способ скрыть одну из его вкладок — это удалить ее из коллекции `TabPages` данного компонента.
4. С помощью свойства `ActiveControl` нельзя перейти к другой вкладке компонента `TabControl`. Для подобного перехода можно использовать либо метод `SelectTab` компонента `TabControl`, либо его свойство `SelectedTab`,

доступное как для чтения, так и для записи. У компонента `TabControl` также имеются два события, связанных с переходом к другой вкладке: `Selecting` и `Selected`.

32.2. Связывание с данными компонентов, не являющихся таблицами

Разместите на вкладке `tabPage2` компонента `tabControl1` метку `label1`. Для этого достаточно после выбора в окне **Toolbox** компонента `Label` щелкнуть мышью где-либо на компоненте `dataGridView2`, занимающем всю область вкладки `tabPage2`. В результате метка будет размещена на вкладке `tabPage2`, так как компонент `dataGridView2` не является контейнером, способным содержать другие компоненты.

Установите свойство `AutoSize` метки `label1` равным **`False`**, а свойство `Dock` равным **`Top`** (свойство `Text`, равное **`label1`**, изменять не требуется). При этом метка `label1` разместится у верхней границы вкладки `tabPage2`, однако будет заслонять верхнюю часть таблицы `dataGridView2`. Чтобы исправить отмеченный недочет, выполните для метки `label1` команду **Send to Back** (нажав кнопку  на панели **Layout** или воспользовавшись контекстным меню метки). После описанных действий вкладка **Оценки** примет вид, приведенный на рис. 32.3.

В конструктор класса `Form1` добавьте оператор

```
label1.DataBindings.Add("Text", bindingSource1, "Surname");
```

Результат. Теперь при переходе на вкладку **Оценки** в ее верхней части отображается фамилия учащегося, оценки которого в данный момент выводятся в таблице.

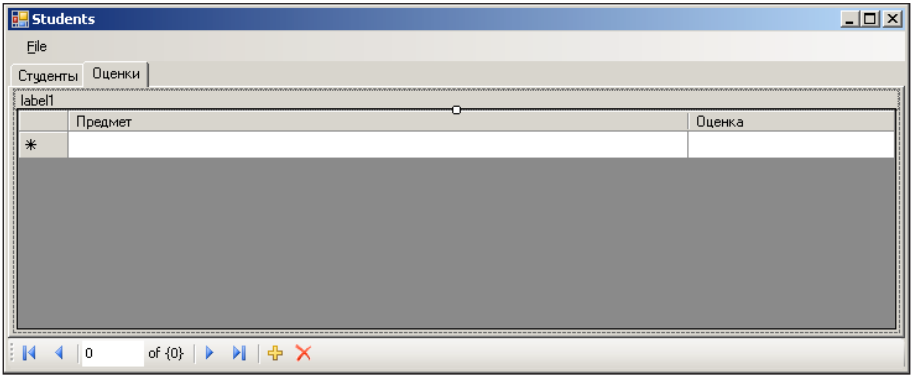


Рис. 32.3. Вид формы `Form1` для проекта STUD2 на промежуточном этапе разработки (вкладка **Оценки**)

Комментарии

1. Свойство-коллекция DataBingings имеется у *всех* визуальных компонентов, поскольку оно определено в общем предке всех визуальных компонентов — классе Control. Это свойство позволяет связать одно или несколько свойств компонента с любыми свойствами других объектов (каждый элемент коллекции DataBindings позволяет установить связь для одного свойства компонента). Для установки связи достаточно вызвать метод Add коллекции DataBindings, указав в качестве первого параметра имя связываемого свойства компонента, в качестве второго параметра — объект, с которым устанавливается связь, а в качестве третьего — имя требуемого свойства указанного объекта. Устанавливаемая связь по умолчанию является *двусторонней*: любое изменение свойства связанного объекта приводит к изменению соответствующего свойства компонента, а изменение свойства компонента приводит к изменению свойства связанного с ним объекта. В программе демонстрируется только одна сторона этого явления, так как пользователь не может изменять текст на метке. Однако если бы вместо метки мы использовали поле ввода textEdit1, то после выполнения следующего оператора (который надо указать в конструкторе формы)

   ```
   textEdit1.DataBindings.Add("Text", bindingSource1, "Surname");
   ```

   любое изменение текста в поле ввода textEdit1 приводило бы к соответствующему изменению свойства Surname текущего элемента набора данных bindingSource1.

2. Для дополнительной настройки связи, устанавливаемой через элементы свойства-коллекции DataBingings, у каждого такого элемента (типа Binding) предусмотрены два свойства: ControlUpdateMode и DataSourceUpdateMode. Первое свойство (перечислимого типа ControlUpdateMode) определяет способ обновления данных в направлении *объект* → *компонент* и может принимать два значения: OnPropertyChanged (обновление свойства компонента при изменении свойства связанного объекта; значение по умолчанию) и Never (отключение автоматического обновления свойства компонента). Второе свойство (перечислимого типа DataSourceUpdateMode) определяет способ обновления данных в направлении *компонент* → *объект* и может принимать три значения: OnValidation (обновление свойства связанного объекта при успешной проверке допустимости соответствующего свойства компонента; значение по умолчанию), OnPropertyChanged (обновление свойства связанного объекта при любом изменении свойства компонента) и Never (отключение автоматического обновления свойства объекта). Если, к примеру, мы связали свойство Surname с компонентом textBox1 (см. ком-

ментарий 1), но не хотим, чтобы изменение текста в этом компоненте приводило к изменению связанного с ним свойства `Surname`, то необходимо в дополнение к оператору, приведенному в конце комментария 1, указать следующий оператор:

```
textBox1.DataBindings[0].DataSourceUpdateMode =
  DataSourceUpdateMode.Never;
```

Заметим, что при отключенном автоматическом обновлении имеется возможность обновить свойства связанных объектов явным образом; для этого предназначены два метода класса `Binding`: `ReadValue` обновляет значение свойства компонента в соответствии с текущим свойством связанного объекта, а `WriteValue` выполняет обратное обновление, т. е. обновляет значение свойства объекта в соответствии с текущим значением свойства компонента.

3. Связь между компонентами и другими объектами можно установить и в режиме дизайна, используя свойство `DataBindings` в окне **Properties**. При разворачивании этого свойства (щелчком мышью на знаке "+" слева от имени свойства) в окне **Properties** отображается список тех свойств текущего компонента, которые чаще всего используются для установки связи с другими объектами. Например, для компонента `TextBox` указываются свойства `Text` и `Tag`. Выбрав требуемое свойство, надо указать связываемый с ним объект и его свойство; действия при указании объекта подобны действиям, выполняемым для указания свойства `DataSource` (см. описание этих действий в *разд. 31.1*). Если требуется установить связь для свойства компонента, которое не указано в списке часто используемых свойств, то надо перейти на строку **(Advanced)** свойства `DataBindings` и нажать кнопку с многоточием  в этой строке. В результате на экране появится диалоговое окно **Formatting and Advanced Binding**, которое позволяет не только установить связь для любого свойства компонента, но и настроить дополнительные характеристики устанавливаемой связи.

4. Описанные в текущем разделе возможности упрощают разработку приложений, в которых наборы данных должны представляться не в виде таблицы, а в виде *заполняемой формы* (бланка), т. е. набора компонентов различного типа (меток, полей ввода, выпадающих списков, изображений и т. п.). Представление данных в виде заполняемой формы, а не в виде таблицы, во многих ситуациях оказывается более удобным (например, если число полей в наборе слишком велико, чтобы объединить все поля в одну экранную таблицу, или если какие-либо поля содержат текст или иллюстрации, размер которых существенно превосходит размер прочих полей).

32.3. Выпадающие списки, связанные с числовыми полями

В наборах данных, которые обрабатываются нашим приложением, имеются три числовых поля со значениями из некоторого небольшого диапазона целых чисел: это поля `Course` (номер курса; значения от 1 до 6, а также 0, если курс не указан), `Group` (номер группы; будем считать, что он может принимать значения от 1 до 20, а также 0, если группа не указана) и `Level` (оценка; значения от 2 до 5, а также 0, если оценка не указана). Чтобы уменьшить число ошибок, которые могут возникать при вводе данных, целесообразно связать эти поля с *выпадающими списками*, содержащими только допустимые значения. Для реализации выпадающих списков, связываемых с *нетекстовыми* полями, предусмотрен специальный механизм, который и будет описан в данном разделе.

Действуя так же, как в *разд. 31.4*, установите для столбцов `course1` и `group1` таблицы `dataGridView1`, а также столбца `level1` таблицы `dataGridView2`, значение свойства `ColumnType` равным **`DataGridViewComboBoxColumn`**.

Создайте новый файл IntData.cs с описанием класса `IntData` (листинг 32.10).

В класс `Form1` добавьте вспомогательный метод `SetIntData` (листинг 32.11) и дополните конструктор этого класса (листинг 32.12).

Листинг 32.10. Текст файла IntData.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace STUD1
{
  class IntData
  {
    private int intValue;
    public int IntValue
    {
      get { return intValue; }
      set { intValue = value; }
    }
    public string StrValue
    {
```

```
      get { return intValue == 0 ? " " : intValue.ToString(); }
      set { int.TryParse(value, out intValue); }
    }
    public IntData(int value)
    {
      intValue = value;
    }
  }
}
```

Листинг 32.11. Метод SetIntData формы Form1

```
private void SetIntData(int a1, int a2, DataGridViewColumn c)
{
  List<IntData> a = new List<IntData>();
  if (a1 < a2)
    for (int i = a1; i <= a2; i++)
      a.Add(new IntData(i));
  else
    for (int i = a1; i >= a2; i--)
      a.Add(new IntData(i));
  a.Add(new IntData(0));
  DataGridViewComboBoxColumn cbc = c as DataGridViewComboBoxColumn;
  cbc.DisplayMember = "StrValue";
  cbc.ValueMember = "IntValue";
  cbc.DataSource = a;
}
```

Листинг 32.12. Добавление к конструктору формы Form1

```
SetIntData(1, 6, dataGridView1.Columns["course1"]);
SetIntData(1, 20, dataGridView1.Columns["group1"]);
SetIntData(5, 2, dataGridView2.Columns["level1"]);
```

Результат. Для ввода данных в поля **Курс**, **Группа** и **Оценка** теперь предусмотрены выпадающие списки; таким образом, ввести недопустимое значение для этих полей невозможно. Кроме допустимых числовых значений в выпадающих

списках предусмотрено "пустое значение", означающее отсутствие информации о значении поля. Для ввода пустого значения достаточно в режиме редактирования нажать клавишу <Пробел>. В наборе данных пустое значение будет храниться в виде числа 0.

Ошибка. При выборе пустого значения для полей **Курс** или **Группа** выход из текущей ячейки блокируется, а в заголовке текущей строки выводится значок , с которым связывается всплывающая подсказка "Строку нельзя преобразовать в число".

Исправление. Откорректируйте метод `dataGridView1_CellValidating`, описанный в *разд. 31.2*, удалив из оператора `switch` все варианты, кроме `case 1` (проверка того, что фамилия студента не является пустой строкой) и `case 3` (проверка правильности ввода даты рождения).

> ***ПРИМЕЧАНИЕ***
>
> Для исправления отмеченной ошибки достаточно удалить из оператора `switch` метода `dataGridView1_CellValidating` варианты, связанные с проверкой свойств `Course` и `Group`, однако модификации таблицы `dataGridView1`, выполненные ранее в *разд. 31.4* и *31.7*, позволяют отказаться и от проверок свойств `Id` и `Gender`.

Комментарии

1. Для возможности работы с нетекстовыми значениями в выпадающем списке предусмотрены три свойства: `DataSource`, `DisplayMember` и `ValueMember`. Два последних свойства содержат имена свойств класса, объекты которого содержатся в коллекции, задаваемой свойством `DataSource`. При этом в `DisplayMember` указывается имя свойства, используемого при *отображении* данных в выпадающем списке, а в `ValueMember` — имя свойства, соответствующего "реальным" данным, для выбора которых предназначен выпадающий список. В нашем случае реальными данными являются целые числа, а их представлениями в выпадающем списке являются либо изображения соответствующих чисел, либо пробел (изображающий число 0); в качестве класса-источника данных используется вспомогательный класс `IntData` (см. листинг 32.10). Для уменьшения размера программного кода повторяющиеся действия оформлены в виде вспомогательного метода `SetIntData` (см. листинг 32.11), параметрами которого являются числа `a1` и `a2`, задающие диапазон значений, добавляемых в список (кроме этого диапазона в список добавляется значение 0, причем оно располагается в списке последним), а также настраиваемый столбец таблицы (необходимо, чтобы этот столбец имел тип `DataGridViewComboBoxColumn`).

2. Описанные в комментарии 1 свойства `DataSource`, `DisplayMember` и `ValueMember` предусмотрены не только для ячеек таблицы с выпадающими списками, но и для обычных списков `ListBox` и выпадающих списков `ComboBox`. Если в компонентах-списках явно задаются эти свойства, то свойство `Items` во внимание не принимается. При использовании этих свойств может оказаться полезным свойство `SelectedValue` (доступное как для чтения, так и для записи), позволяющее определить "реальное" значение, соответствующее текущему пункту списка.

32.4. Вычисляемые столбцы

В таблицах, содержащих наборы данных, часто требуется отображать информацию, вычисляемую по имеющимся данным. Для отображения подобной информации в компоненте `DataGridView` обычно используются столбцы, не привязанные к источнику данных (unbound columns).

В описание класса `Student`, содержащееся в файле Student.cs, добавьте новое свойство `AvrLevel`, доступное только для чтения (листинг 32.13).

Листинг 32.13. Добавление к описанию класса `Student` из файла Student.cs

```
public double AvrLevel
{
  get
  {
    if (Marks == null)
      return 0;
    double sum = 0;
    int cnt = 0;
    foreach (Mark m in Marks)
      if (m.Level != 0)
      {
        sum += m.Level;
        cnt++;
      }
    if (cnt > 0)
      sum /= cnt;
    return sum;
  }
}
```

Выделите компонент dataGridView1, щелкните мышью на значке ▶ около правого верхнего угла компонента и выберите в появившемся вспомогательном окне пункт **Add Column...**. В появившемся окне **Add Column** выберите вариант **Unbound column** и установите для него следующие значения:

Name: **avrLevel1**

Type: **DataGridViewTextBoxColumn**

HeaderText: **Средний балл**

Кроме того, установите флажок **Read Only** (таким образом, окно должно содержать два установленных флажка: **Visible** и **Read Only**). Нажмите кнопку **Add**; при этом в таблицу dataGridView1 будет добавлен новый столбец с указанными свойствами. После этого закройте окно **Add Column**, нажав кнопку **Close** или клавишу <Esc>. В результате выполнения описанных действий таблица dataGridView1 примет вид, приведенный на рис. 32.4.

Определите обработчик события CellFormatting для таблицы dataGridView1 (листинг 32.14).

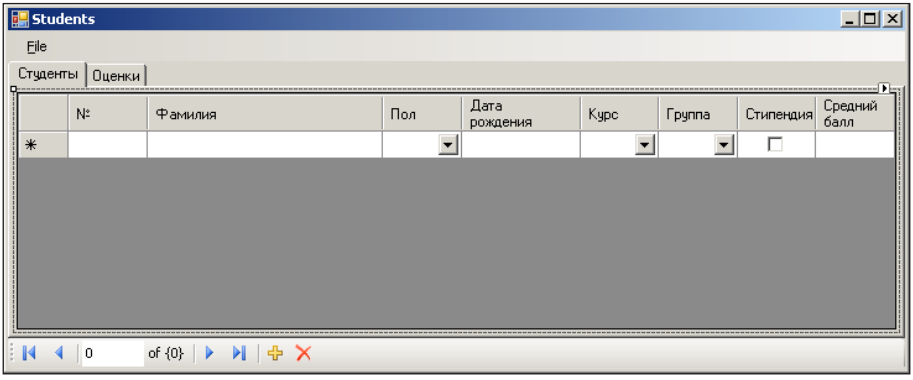


Рис. 32.4. Вид формы Form1 для проекта STUD2 на промежуточном этапе разработки (вкладка **Студенты**)

Листинг 32.14. Обработчик dataGridView1.CellFormatting

```
private void dataGridView1_CellFormatting(object sender,
  DataGridViewCellFormattingEventArgs e)
{
  if (dataGridView1.Rows[e.RowIndex].IsNewRow ||
    e.ColumnIndex != dataGridView1.Columns["avrLevel1"].Index)
    return;
```

```
  e.FormattingApplied = true;
  double a = (bindingSource1[e.RowIndex] as Student).AvrLevel;
  e.Value = a == 0 ? "" : a.ToString("f2");
}
```

Результат. Если для студента определена хотя бы одна оценка, то в столбце **Средний балл** для него отображается средний балл по всем полученным оценкам (если для текущего студента оценки не указаны, то столбец **Средний балл** остается пустым). Столбец **Средний балл** доступен только для чтения.

Комментарии

1. Пересчет среднего балла производится при наступлении события `CellFormatting`, которое возникает при отображении элемента данных в ячейке таблицы. Обработчик события `CellFormatting` позволяет самому пользователю определить требуемое значение в ячейке, указав его в свойстве `e.Value` (см. листинг 32.14). При явном задании значения необходимо также положить свойство `e.FormattingApplied` равным `true`; в этом случае компонент `DataGridView` не будет выполнять дополнительные стандартные действия по форматированию ячейки.
2. Для того чтобы явным образом "заставить" таблицу `DataGridView` обновить свои данные, отображаемые на экране, достаточно вызвать ее метод `Refresh` (без параметров). В нашей программе это не требуется, поскольку необходимость в пересчете среднего балла возникает только при изменении таблицы **Оценки**, а в этот момент таблица **Студенты** на экране не отображается. Заметим, что при отображении на экране ранее скрытой таблицы обновление ее данных выполняется автоматически.

32.5. Сортировка данных

Используя конструктор меню *(см. разд. 18.1)*, создайте в компоненте `menuStrip1` новый пункт меню первого уровня с текстом **`&Data`** и с помощью окна **Properties** измените имя этого пункта (т. е. свойство `Name`) на **`data1`**. В выпадающем меню, связанном с пунктом **Data**, создайте пункт с текстом **`&Sort by...`** и измените имя этого пункта на **`sortBy1`**. Затем перейдите в заготовку меню *третьего* уровня, связанную с пунктом **Sort by...**, добавьте в нее четыре пункта меню с текстом **`&Id`**, **`&Surname`**, **`&Course+Group`** и **`&Average Level`** (рис. 32.5) и измените имена этих пунктов на **`id2`**, **`surname2`**, **`course2`** и **`avrLevel2`** соответственно.

В конструктор класса `Form1` добавьте новые операторы (листинг 32.15).

В описание класса `Form1` добавьте четыре вспомогательных метода: `CompareById`, `CompareBySurname`, `CompareByCourse`, `CompareByAvrLevel` (листинг 32.16).

Определите обработчик события `Click` для пункта меню `id2` (листинг 32.17), после чего свяжите созданный обработчик с событием `Click` пунктов меню `surname2`, `course2` и `avrLevel2`.

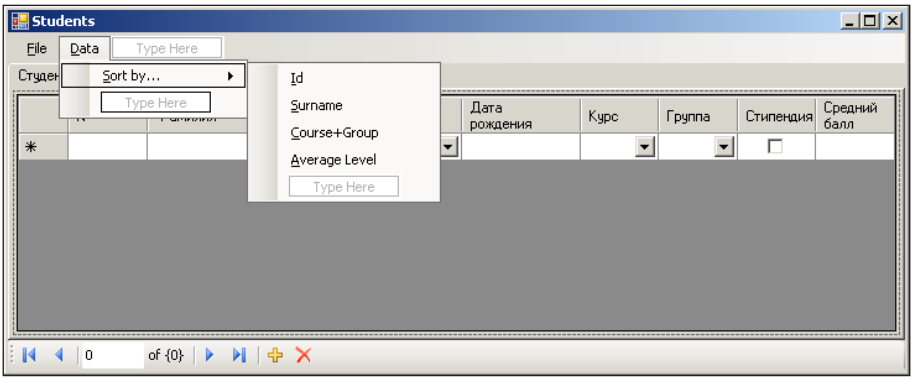


Рис. 32.5. Вид формы `Form1` для проекта STUD2 после добавления группы меню **Data**

Листинг 32.15. Добавление к конструктору формы `Form1`

```
id2.Tag = 0;
surname2.Tag = 1;
course2.Tag = 2;
avrLevel2.Tag = 3;
```

Листинг 32.16. Методы `CompareById`, `CompareBySurname`, `CompareByCourse`, `CompareByAvrLevel` формы `Form1`

```
private int CompareById(Student a, Student b)
{
  return a.Id - b.Id;
}
private int CompareBySurname(Student a, Student b)
{
  return a.Surname.CompareTo(b.Surname);
}
private int CompareByCourse(Student a, Student b)
{
```

```
  int res = a.Course - b.Course;
  if (res == 0)
    res = a.Group - b.Group;
  if (res == 0)
    res = a.Surname.CompareTo(b.Surname);
  return res;
}
private int CompareByAvrLevel(Student a, Student b)
{
  int res = Math.Sign(b.AvrLevel - a.AvrLevel);
  if (res == 0)
    res = a.Surname.CompareTo(b.Surname);
  return res;
}
```

Листинг 32.17. Обработчик `id2.Click`

```
private void id2_Click(object sender, EventArgs e)
{
  if (dataGridView1.RowCount == 1)
    return;
  dataGridView1.CurrentCell = dataGridView1[0, 0];
  Comparison<Student> comp = CompareById;
  switch ((int)(sender as ToolStripMenuItem).Tag)
  {
    case 1:
      comp = CompareBySurname;
      break;
    case 2:
      comp = CompareByCourse;
      break;
    case 3:
      comp = CompareByAvrLevel;
      break;
  }
  (bindingSource1.DataSource as List<Student>).Sort(comp);
  bindingSource1.ResetBindings(false);
}
```

Результат. Добавленная в меню группа команд **Sort by...** позволяет сортировать набор данных **Студенты** по четырем различным ключам: полю **№**, полю **Фамилия**, комбинации полей **Курс** и **Группа** (если у нескольких студентов номера курса и группы являются одинаковыми, то они сортируются по фамилиям) и вычисляемому полю **Средний балл**. В последнем случае данные сортируются по *убыванию* среднего балла; при равных средних баллах студенты сортируются по фамилиям, если для студента не указано ни одной оценки, то средний балл считается равным нулю. В результате сортировки в таблице `dataGridView1` изменяется порядок строк; текущей ячейкой становится первая ячейка первой строки. При записи данных в XML-файл сохраняется их порядок, полученный в результате последней сортировки.

Комментарии

1. Для сортировки данных, содержащихся в обобщенном списке `List<T>`, предусмотрено несколько перегруженных методов `Sort`. Мы воспользовались вариантом метода `Sort(comparison)`, в котором указывается *функция сравнения* `comparison`, используемая при сортировке. Функция сравнения содержит два сравниваемых параметра `a` и `b` типа `T` и возвращает значение типа `int` (отрицательное, если `a` меньше `b`, нулевое, если `a` равно `b`, и положительное, если `a` больше `b`). В программе определены четыре функции сравнения (см. листинг 32.16). При выборе одного из пунктов меню, связанных с сортировкой, вызывается метод `Sort`, в который передается функция, соответствующая выбранному пункту меню (см. листинг 32.17). Заметим, что параметр `comparison` метода `Sort` имеет тип *делегата-обобщения* `Comparison<T>`.

   Номер варианта сортировки для каждого пункта меню хранится в его свойстве `Tag` (см. листинг 32.15); это позволяет использовать для всех пунктов меню один и тот же обработчик `id2_Click`.

2. После завершения сортировки необходимо обновить компоненты, связанные с отсортированным набором данных `BindingSource` (в нашем случае надо обновить таблицу `dataGridView1`, связанную с набором данных `bindingSource1`). Для этого достаточно вызвать метод `ResetBindings` набора данных (см. листинг 32.17). Если в наборе данных изменились только значения, но не структура, то параметр метода `ResetBindings` следует положить равным `false`; это увеличит скорость его выполнения.

32.6. Поиск по шаблону

Подключите к проекту STUD2 библиотеку Microsoft.VisualBasic. Для этого щелкните правой кнопкой мыши на строке **References** в окне **Solution Explorer**,

выберите из появившегося контекстного меню команду **Add Reference...** и в появившемся окне **Add Reference** выделите пункт **Microsoft.VisualBasic**, после чего нажмите кнопку **OK**.

В начало файла Form1.cs добавьте оператор

```
using Microsoft.VisualBasic;
```

В описание класса `Form1` добавьте новое поле:

```
private string surnameToFind = "";
```

Дополните выпадающее меню, связанное с пунктом **Data** *(см. разд. 32.5)*, добавив в него пункт с текстом **`&Find...`**, и измените имя добавленного пункта меню на **`find1`**.

Измените первый оператор метода `dataGridView1_RowEnter` (см. листинг 32.6):

```
find1.Enabled = bindingNavigatorDeleteItem.Enabled =
  !dataGridView1.Rows[e.RowIndex].IsNewRow;
```

Аналогичным образом измените первый оператор метода `dataGridView1_CurrentCellDirtyStateChanged` (см. листинги 31.19 и 32.8):

```
find1.Enabled = menuStrip1.Enabled = !dataGridView1.IsCurrentCellDirty;
```

Определите обработчик события `Click` для пункта меню `find1` (листинг 32.18).

Листинг 32.18. Обработчик `find1.Click`

```
private void find1_Click(object sender, EventArgs e)
{
  surnameToFind =
    Interaction.InputBox("Введите начальную часть фамилии для поиска:",
    "Поиск по фамилии", surnameToFind, -1, -1).Trim();
  if (surnameToFind == "")
    return;
  int ind = (bindingSource1.DataSource as
    List<Student>).FindIndex(dataGridView1.CurrentRow.Index,
    delegate(Student a)
    {
      return a.Surname.StartsWith(surnameToFind,
        StringComparison.OrdinalIgnoreCase);
    });
```

```
  if (ind != -1)
    dataGridView1.CurrentCell = dataGridView1[1, ind];
  else
    MessageBox.Show("Фамилия не найдена", "Поиск по фамилии");
}
```

Результат. Добавленная в меню команда **Find...** позволяет организовать поиск студента по начальной части его фамилии. Для ввода требуемой начальной части фамилии используется стандартное диалоговое окно (рис. 32.6). При отображении этого окна в нем выводится предыдущий текст для поиска. В случае, если введенный текст является пустым или содержит только пробелы, а также если диалоговое окно закрывается нажатием кнопки **Cancel** или клавиши <Esc>, действие команды **Find** отменяется, а текст для поиска полагается равным пустой строке.

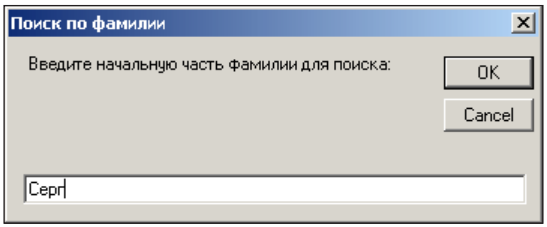


Рис. 32.6. Вид стандартного диалогового окна для ввода текстовой информации

Поиск начинается с текущего элемента набора данных. Если требуемая фамилия найдена, то ячейка таблицы с найденной фамилией делается текущей; если фамилия не найдена, то об этом выводится сообщение. Поиск выполняется без учета регистра символов; кроме того, в тексте для поиска удаляются начальные и конечные пробелы.

Если текущей является последняя строка таблицы (строка NewRow), то команда **Find** недоступна.

> ***ПРИМЕЧАНИЕ***
>
> Для отображения на экране диалогового окна для ввода строковых данных мы воспользовались методом `InputBox` класса `Interaction` из пространства имен `Microsoft.VisualBasic` (см. также комментарий 2 в *разд. 5.6*).

Приведем изображение работающей программы после сортировки данных по среднему баллу (рис. 32.7).

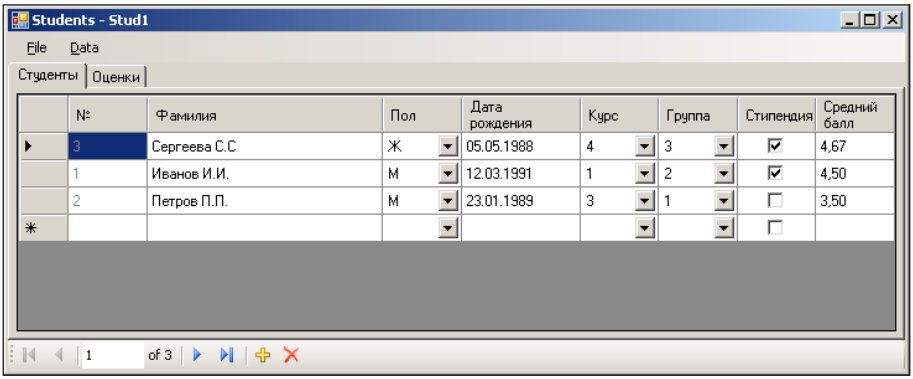


Рис. 32.7. Вид работающего приложения STUD2

Комментарии

1. Поиск организуется с помощью одного из перегруженных вариантов метода `FindIndex` обобщенного класса `List<T>`, а именно — варианта с двумя параметрами: номера элемента, с которого начинается поиск, и *тестовой функции*. Тестовая функция должна иметь один параметр типа `T` и возвращать логическое значение `true`, если ее параметр удовлетворяет критерию поиска, и `false` в противном случае. Параметр метода `FindIndex`, соответствующий тестовой функции, описывается в виде *делегата-обобщения* `Predicate<T>`. Обратите внимание на то, что в нашей программе (см. листинг 32.18) мы создаем тестовую функцию-делегата "на лету", указывая ее описание в качестве второго параметра метода `FindIndex` (заголовком в описании служит ключевое слово `delegate`, снабженное списком параметров). Методы, определенные таким образом, называются *анонимными* (anonymous methods). Возможность описания анонимных методов непосредственно при их передаче в качестве параметров появилась в .NET Framework 2.0. Заметим, что можно также создавать анонимные обработчики событий, помещая их описания непосредственно в оператор подключения обработчика к соответствующему событию. Если, например, требуется, чтобы при нажатии кнопки `button1` происходило закрытие формы `Form1` *(см. разд. 4.2)*, то достаточно добавить в конструктор формы `Form1` следующий оператор:

```
button1.Click += delegate { Close(); };
```

Приведенный оператор демонстрирует еще одну особенность, связанную с созданием анонимных методов (в частности, обработчиков): если в подобном методе не требуется использовать его параметры, то список параметров после слова `delegate` можно не указывать.

2. Для проверки того, что начальная часть строки совпадает с требуемой подстрокой, предназначен метод `StartsWith` класса `string`. В обработчике `find1_Click` мы использовали вариант этого метода с дополнительным параметром перечислимого типа `StringComparison`, который определяет режим поиска. Значение `StringComparison.OrdinalIgnoreCase`, указанное при вызове метода `StartsWith`, обеспечивает режим поиска без учета регистра букв. В этом режиме, как и в режиме `StringComparison.Ordinal`, обеспечивающем поиск с учетом регистра, символы сравниваются по их *кодам*. Имеются также варианты режима поиска, в которых при сравнении символов учитываются особенности региональных настроек, однако эти варианты работают медленнее, не давая в ситуациях, подобных нашей, никаких дополнительных преимуществ. "Парным" к методу `StartsWith` является метод `EndsWith`, позволяющий выяснить, оканчивается ли строка требуемой подстрокой. Заметим, что перечислимый тип `StringComparison` и варианты методов `StartsWith` и `EndsWith` с параметром этого типа появились в .NET Framework 2.0.

   Для удаления начальных и конечных пробелов из строки, содержащей текст для поиска, был использован метод `Trim` класса `string`.

Глава 33

Создание компонентов во время выполнения программы: проект HTOWERS

Завершающий пример книги, проект HTOWERS, представляет собой компьютерную реализацию известной логической задачи "Ханойские башни". Эта задача состоит в следующем: имеются три колышка, на один из которых нанизано (в порядке уменьшения размера) несколько дисков, образующих "башню"; требуется переместить всю башню на один из пустых колышков, пользуясь другим пустым колышком как вспомогательным. Переносить можно по одному диску, причем больший диск нельзя помещать на меньший. Заметим, что для решения задачи с N дисками минимальное число переносов равно $2^N - 1$ (см. [2]).

В приложении HTOWERS вместо колышков используются компоненты-контейнеры `GroupBox`, а вместо дисков перемещаются прямоугольные блоки (компоненты `Label`), создаваемые непосредственно в ходе выполнения программы. Перемещение блоков реализуется с помощью механизма drag & drop. Предусматриваются средства проверки правильности решения, а также демо-режим, показывающий, в каком порядке надо перемещать блоки, чтобы решить задачу. В заключительном разделе главы описываются действия, позволяющие задать идентификационные данные для разработанного приложения (название приложения, его краткое описание, сведения об авторе, номер версии).

33.1. Создание начальной позиции

После создания проекта HTOWERS разместите в форме `Form1` три компонента типа `GroupBox` (они получат имена `groupBox1`—`groupBox3`) и компонент типа `NumericUpDown` (он получит имя `numericUpDown1`). Настройте свойства формы `Form1` и добавленных компонентов (листинг 33.1) и расположите компоненты в соответствии с рис. 33.1.

Определите обработчик события `Load` для формы `Form1` (листинг 33.2).

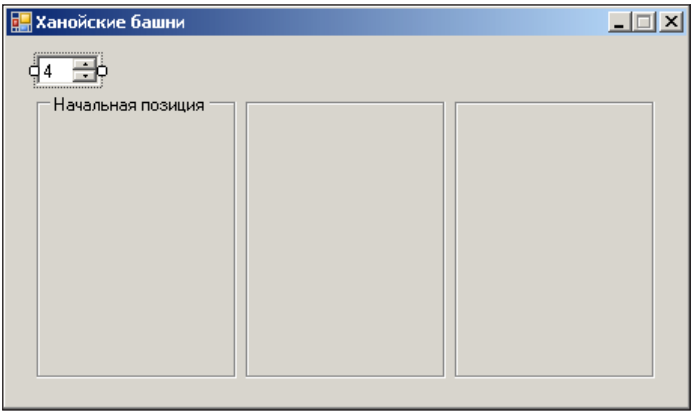


Рис. 33.1. Вид формы `Form1` для проекта HTOWERS на начальном этапе разработки

Листинг 33.1. Настройка свойств

```
Form1: Text = Ханойские башни, MaximizeBox = False,
  FormBorderStyle = FixedSingle, StartPosition = CenterScreen
groupBox1: Text = Начальная позиция
groupBox2–groupBox3: Text = пустая строка
numericUpDown1: Minimum = 2, Maximum = 10, Value = 4
```

Листинг 33.2. Обработчик `Form1.Load`

```
private void Form1_Load(object sender, EventArgs e)
{
  Random r = new Random();
  int n = (int)numericUpDown1.Value,
    w = groupBox1.Width,
    h = groupBox1.Height;
  for (int i = 0; i < n; i++)
  {
    Label lb = new Label();
    lb.Parent = groupBox1;
    lb.BorderStyle = BorderStyle.FixedSingle;
    lb.Size = new Size((w - 10) * (n - i) / n, (h - 15) / n);
    lb.Location =
```

```
      new Point((w - lb.Width) / 2, h - 2 - (i + 1) * lb.Height);
    lb.BackColor = Color.FromArgb(r.Next(256), r.Next(256), r.Next(256));
  }
}
```

Результат. При запуске программы на компоненте `groupBox1` рисуется башня из четырех разноцветных блоков — компонентов типа `Label` (цвет блоков определяется случайным образом).

Комментарий

Компоненты-метки `Label` создаются непосредственно при выполнении метода `Form1_Load`. После создания метки (с помощью конструктора `new Label()`) для нее определяется родительский компонент `Parent`, в котором будет изображена созданная метка, стиль ее границы, размер, положение и фоновый цвет.

В качестве родительского компонента, определяемого свойством `Parent`, можно указать любой компонент-контейнер (в частности, форму). Вместо оператора `lb.Parent = groupBox1` можно использовать эквивалентный ему вызов метода `Add` для свойства-коллекции `Controls` родительского компонента:

```
groupBox1.Controls.Add(lb);
```

Родительский компонент отвечает за перерисовку всех своих дочерних компонентов. Если компонент не имеет родителя, то его невозможно отобразить на экране.

33.2. Перерисовка башни при изменении количества блоков

Определите обработчик события `ValueChanged` для компонента `numericUpDown1` (листинг 33.3).

Листинг 33.3. Обработчик `numericUpDown1.ValueChanged`

```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
{
  Form1_Load(this, null);
}
```

Результат. При изменении значения компонента `numericUpDown1` создается новая башня с указанным числом блоков.

Ошибка. Новая башня рисуется *под* старой.

Исправление. В класс `Form1` добавьте новый метод `label_Dispose` (листинг 33.4).

В *начало* метода `numericUpDown1_ValueChanged` вставьте оператор

```
label_Dispose(groupBox1);
```

Листинг 33.4. Метод `label_Dispose` формы `Form1`

```
private void label_Dispose(GroupBox gb)
{
  for (int i = gb.Controls.Count - 1; i >= 0; i--)
    gb.Controls[i].Dispose();
}
```

Результат. Теперь все блоки старой башни исчезают с экрана, а также освобождают свои ресурсы (благодаря вызову метода `Dispose`). См. также комментарий 1.

Недочет. В процессе формирования новой башни на экране (в верхней части компонента `groupBox1`) мелькают посторонние фрагменты изображений. Отмеченный недочет связан с тем, что после определения родительского компонента метка *немедленно* рисуется на нем, а все последующие изменения ее свойств приводят к ее перерисовке.

Исправление. В методе `Form1_Load` (см. листинг 33.2) переместите оператор

```
lb.Parent = groupBox1;
```

в конец цикла (после оператора, определяющего фоновый цвет метки).

Результат. Теперь все настройки свойств метки выполняются до определения ее родительского компонента, и поэтому не сопровождаются ее перерисовкой на экране (метка рисуется единственный раз после задания всех ее свойств). См. также комментарий 2.

Комментарии

1. В коллекции `Controls`, имеющейся у любого визуального компонента, предусмотрен метод `Clear`, который позволяет удалить из нее все элементы. В результате выполнения метода `Clear` для компонента-контейнера все его дочерние компоненты перестают изображаться на экране. Однако при этом дочерние компоненты не уничтожаются, т. е. для них не вызывается метод `Dispose` (в этом можно убедиться, сохранив один из дочерних компонентов

во вспомогательной переменной и попытавшись уже после вызова метода `Clear` отобразить этот компонент на экране: компонент действительно будет нарисован). Поэтому нам пришлось реализовать дополнительный метод (см. листинг 33.4), который обеспечивает разрушение дочерних компонентов-меток для указанного контейнера типа `GroupBox`.

2. В документации по .NET сказано, что при добавлении в контейнер большого количества визуальных компонентов следует предварительно вызвать метод `SuspendLayout` данного контейнера, а после завершения добавления вызвать метод `ResumeLayout` (утверждается, что при вызове метода `SuspendLayout` перерисовка на контейнере блокируется). К сожалению, попытка исправить указанный выше недочет, не перемещая оператор со свойством `Parent` на новое место, но добавляя перед циклом вызов `groupBox1.SuspendLayout()`, а после цикла — вызов `groupBox1.ResumeLayout()`, не привела к успеху: посторонние изображения продолжали мелькать на экране.

33.3. Перетаскивание блоков на новое место

В конструктор класса `Form1` добавьте новый оператор:

```
groupBox1.AllowDrop = groupBox2.AllowDrop = groupBox3.AllowDrop = true;
```

В класс `Form1` добавьте новые методы `label_MouseDown` и `label_Move` (листинг 33.5).

В методе `Form1_Load` в конец тела цикла `for` добавьте следующий оператор:

```
lb.MouseDown += label_MouseDown;
```

Определите обработчики событий `DragEnter` и `DragDrop` для компонента `groupBox1` (листинг 33.6), после чего свяжите созданные обработчики с событиями `DragEnter` и `DragDrop` компонентов `groupBox2` и `groupBox3`.

Листинг 33.5. Методы `label_MouseDown` и `label_Move` формы `Form1`

```
private void label_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
    DoDragDrop(sender, DragDropEffects.Move);
}
private void label_Move(Label lb, GroupBox gb)
{
  lb.Parent = gb;
  lb.Top = gb.Height - 2 - gb.Controls.Count * lb.Height;
}
```

Листинг 33.6. Обработчики `groupBox1.DragEnter` и `groupBox1.DragDrop`

```
private void groupBox1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
}
private void groupBox1_DragDrop(object sender, DragEventArgs e)
{
  Label lb = e.Data.GetData(typeof(Label)) as Label;
  GroupBox gb = sender as GroupBox;
  if (gb == lb.Parent)
    return;
  label_Move(lb, gb);
}
```

Результат. Любой блок (т. е. метку типа `Label`) можно переместить на другую башню (т. е. в другой компонент `GroupBox`), причем перемещенный блок всегда будет располагаться на вершине башни.

> ***ПРИМЕЧАНИЕ***
>
> Действия, связанные с перемещением метки в другой компонент `GroupBox`, оформлены в виде отдельного метода `label_Move` (см. листинг 33.5), поскольку в дальнейшем они будут выполняться не только при наступлении события `DragDrop`, но и в демонстрационном режиме программы *(см. разд. 33.7)*.

Ошибка. Переместить можно не только верхний, но и любой другой блок башни.

Исправление. Измените метод `groupBox1_DragEnter` (листинг 33.7).

Листинг 33.7. Первая корректировка метода `groupBox1_DragEnter`

```
private void groupBox1_DragEnter(object sender, DragEventArgs e)
{
  Label lb = e.Data.GetData(typeof(Label)) as Label;
  if (lb.Parent.Controls[lb.Parent.Controls.Count - 1] != lb)
    e.Effect = DragDropEffects.None;
  else
    e.Effect = DragDropEffects.Move;
}
```

Результат. Теперь переместить можно только верхний блок башни; при попытке переместить нижние блоки курсор перетаскивания становится запрещающим.

> ***ПРИМЕЧАНИЕ***
>
> Мы воспользовались тем обстоятельством, что при добавлении дочернего компонента в коллекцию `Controls` родительского компонента добавленный компонент располагается в ее конце; таким образом, верхний элемент башни всегда имеет в коллекции `Controls` индекс, равный числу элементов коллекции `Controls.Count` минус `1`.

Осталось учесть дополнительное условие задачи: блок не может перемещаться на башню с верхним блоком меньшего размера. Для этого еще раз изменим метод `groupBox1_DragEnter` (листинг 33.8).

Листинг 33.8. Вторая корректировка метода `groupBox1_DragEnter`

```
private void groupBox1_DragEnter(object sender, DragEventArgs e)
{
  Label lb = e.Data.GetData(typeof(Label)) as Label;
  int k = int.MaxValue;
  GroupBox gb = sender as GroupBox;
  if (gb.Controls.Count > 0)
    k = gb.Controls[gb.Controls.Count - 1].Width;
  if (lb.Parent.Controls[lb.Parent.Controls.Count - 1] != lb
    || lb.Width > k)
    e.Effect = DragDropEffects.None;
  else
    e.Effect = DragDropEffects.Move;
}
```

Результат. При перемещении блока учитывается дополнительное условие.

> ***ПРИМЕЧАНИЕ***
>
> В последнем варианте метода `groupBox1_DragEnter` используется переменная `k`, в которую записывается ширина верхнего блока компонента-приемника или максимальное возможное значение типа `int` (равное `int.MaxValue`), если компонент-приемник не содержит блоков.

Ошибка. При изменении количества меток с помощью компонента `numericUpDown1` удаляются только те метки, которые находятся в компоненте `groupBox1`.

Исправление. Измените метод numericUpDown1_ValueChanged (листинг 33.9).

Листинг 33.9. Новый вариант метода numericUpDown1_ValueChanged

```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
{
  label_Dispose(groupBox1);
  label_Dispose(groupBox2);
  label_Dispose(groupBox3);
  Form1_Load(this, null);
}
```

Комментарий

Режим перетаскивания drag & drop был подробно рассмотрен в *главе 11*. Отметим, что новое значение свойства AllowDrop для компонентов GroupBox пришлось указывать в конструкторе формы, так как это свойство не отображается в окне **Properties**. Далее, обратите внимание на то, что для меток Label свойство AllowDrop осталось равным false (значение по умолчанию), поэтому метки не могут выступать в роли приемников при перетаскивании (они "невидимы" для режима перетаскивания). Это обстоятельство проявляется в том, что курсор перетаскивания над метками не принимает запрещающий вид, а если отпустить объект-источник над меткой, то событие DragDrop возникнет не для метки, а для того компонента-контейнера GroupBox, который содержит эту метку.

33.4. Восстановление начальной позиции и подсчет числа перемещений блоков

Разместите в форме Form1 кнопку button1, свойству Text кнопки присвойте значение **Сброс** и свяжите событие Click кнопки button1 с существующим обработчиком numericUpDown1_ValueChanged. Кроме того, разместите в форме Form1 метку label1 (ее свойство Text изменять не требуется). Расположите добавленные компоненты в соответствии с рис. 33.2.

В класс Form1 добавьте описания двух полей

```
private int count;
private int minCount;
```

и вспомогательный метод Info (листинг 33.10).

В метод `Form1_Load` добавьте три оператора:

```
count = 0;
minCount = (int)Math.Round(Math.Pow(2, n)) - 1;
Info();
```

В метод `label_Move` добавьте два оператора:

```
count++;
Info();
```

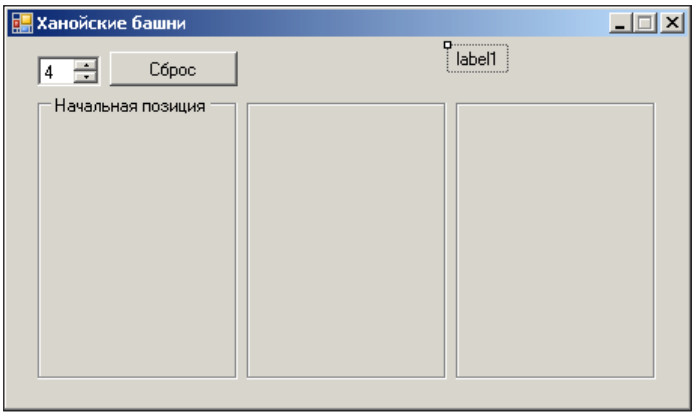


Рис. 33.2. Вид формы `Form1` для проекта HTOWERS на промежуточном этапе разработки

Листинг 33.10. Метод `Info` формы `Form1`

```
private void Info()
{
  label1.Text = string.Format("Число ходов: {0} ({1})", count, minCount);
}
```

Результат. Для восстановления начальной позиции с тем же количеством блоков следует нажать кнопку **Сброс**. Если при восстановлении начальной позиции требуется изменить число блоков, то по-прежнему достаточно указать новое значение в компоненте `numericUpDown1` (нажимать кнопку **Сброс** в этом случае не требуется).

Информация о числе ходов (т. е. перемещений блоков) выводится в тексте метки `label1`. Там же в скобках указывается количество ходов, минимально необходимое для решения задачи с данным числом блоков n (это количество равно $2^n - 1$).

ПРИМЕЧАНИЕ

Обратите внимание на то, что перемещения блока в пределах одного и того же компонента `GroupBox` при подсчете числа ходов не учитываются.

Комментарий

Для нахождения величины 2^n была использована функция `Pow` класса `Math`. Поскольку она возвращает результат типа `double`, полученное значение должно преобразовываться к целому типу. Так как при преобразовании `(int)` происходит отбрасывание дробной части, мы "на всякий случай" предварительно выполняем округление полученного числа до ближайшего целого с помощью функции `Round`. Заметим, что после округления необходимость в преобразовании `(int)` сохраняется, так как функция `Round` возвращает значение типа `double` (хотя и с нулевой дробной частью).

33.5. Проверка решения задачи

Разместите в форме `Form1` под имеющейся меткой `label1` еще одну метку (`label2`), ее свойству `Text` присвойте значение **`Задача решена!`**, а свойству `ForeColor` — значение **`Green`**.

В метод `Form1_Load` добавьте оператор:

```
label2.Visible = false;
```

Добавьте в конец метода `label_Move` следующий фрагмент:

```
if (groupBox2.Controls.Count == numericUpDown1.Value
  || groupBox3.Controls.Count == numericUpDown1.Value)
  label2.Visible = true;
```

Результат. Задача считается решенной и об этом выводится сообщение "Задача решена!", если размер башни в конечной позиции (т. е. в компоненте `groupBox2` или `groupBox3`) равен общему числу блоков.

Недочет. После решения задачи по-прежнему разрешено перемещение блоков.

Исправление. Добавьте в начало метода `groupBox1_DragEnter` (см. листинг 33.8) следующий фрагмент:

```
if (label2.Visible)
{
  e.Effect = DragDropEffects.None;
  return;
}
```

Результат. После решения задачи блоки нельзя перемещать.

33.6. Выполнение задачи в демо-режиме

Разместите в форме `Form1` еще одну кнопку (`button2`) и присвойте ее свойству `Text` значение **`Демо-режим`**. Форма `Form1` примет вид, приведенный на рис. 33.3.

В класс `Form1` добавьте новый метод `Step` (листинг 33.11) и определите обработчик события `Click` для кнопки `button2` (листинг 33.12).

В начало метода `label_MouseDown` добавьте следующий фрагмент:

```
if (!button1.Enabled)
  return;
```

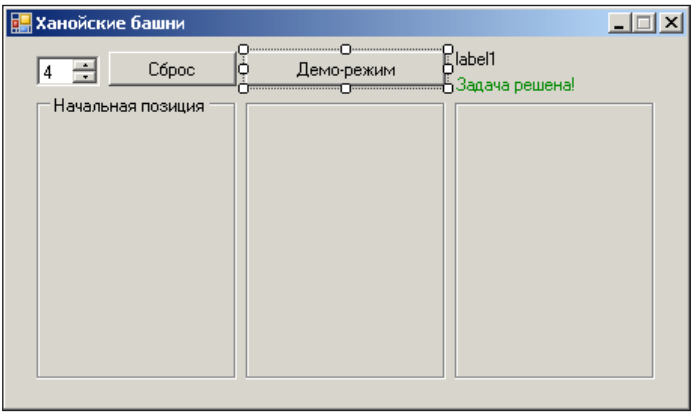


Рис. 33.3. Окончательный вид формы `Form1` для проекта HTOWERS

Листинг 33.11. Метод `Step` формы `Form1`

```
private void Step(int n, GroupBox src, GroupBox dst, GroupBox tmp)
{
  if (n == 0)
    return;
  Step(n - 1, src, tmp, dst);
  if (button1.Enabled)
    return;
  label_Move(src.Controls[src.Controls.Count - 1] as Label, dst);
  Application.DoEvents();
  System.Threading.Thread.Sleep(1500 / ((int)numericUpDown1.Value) - 1);
  Step(n - 1, tmp, dst, src);
}
```

Листинг 33.12. Обработчик `button2.Click`

```
private void button2_Click(object sender, EventArgs e)
{
  numericUpDown1.Enabled = button1.Enabled = !button1.Enabled;
  if (!button1.Enabled)
  {
    if (groupBox1.Controls.Count != numericUpDown1.Value)
      numericUpDown1_ValueChanged(null, null);
    Step((int)numericUpDown1.Value, groupBox1, groupBox3, groupBox2);
    numericUpDown1.Enabled = button1.Enabled = true;
  }
}
```

Результат. При нажатии кнопки **Демо-режим** программа переходит в *демонстрационный режим*, в котором показывается правильное решение задачи с указанным числом блоков (блоки перемещаются автоматически). В демо-режиме блокируются компонент `numericUpDown1` и кнопка **Сброс**; кроме того, в нем запрещено перетаскивание меток. Выход из демо-режима происходит после завершения решения задачи, а также при повторном нажатии кнопки **Демо-режим** (в последнем случае после выхода из демо-режима можно продолжать решать задачу самостоятельно).

Ошибка. При попытке завершить программу, находящуюся в демо-режиме, возникает ошибка времени выполнения `ArgumentOutOfRangeException`, связанная с тем, что ранее вызванные методы `Step` продолжают выполняться уже после того, как свойства-коллекции `Controls` компонентов `GroupBox` были очищены в результате завершающих действий программы.

Исправление. Определите обработчик события `FormClosed` для формы `Form1` (листинг 33.13).

Листинг 33.13. Обработчик `Form1.FormClosed`

```
private void Form1_FormClosed(object sender, FormClosedEventArgs e)
{
  button1.Enabled = true;
}
```

Результат. Теперь при закрытии формы кнопка `button1` делается доступной, что позволяет немедленно завершить все рекурсивные вызовы метода `Step` без обращения к коллекциям `Controls`.

Приведем изображение работающей программы после решения задачи с пятью блоками (рис. 33.4).

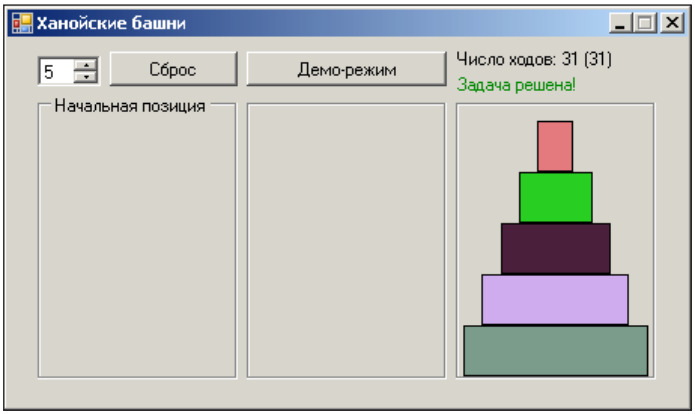


Рис. 33.4. Вид работающего приложения HTOWERS

Комментарий

Метод `Step(n, src, dst, tmp)` определяет действия, необходимые для решения задачи с *n* блоками; при этом параметры *src*, *dst* и *tmp* типа `GroupBox` определяют начальную позицию башни *src* (source), конечную позицию *dst* (destination) и вспомогательную область *tmp* (temporary), используемую при перемещении блоков. Очевидно, что для решения задачи с *n* блоками достаточно выполнить три действия:

1. Переместить башню, состоящую из *n* – 1 блока, из области *src* в область *tmp*, используя область *dst* как вспомогательную.
2. Переместить нижний, *n*-й блок из области *src* в область *dst*.
3. Переместить башню, состоящую из *n* – 1 блока, из области *tmp* в область *dst*, используя область *src* как вспомогательную.

При этом действия 1 и 3 сводятся к вызову того же метода `Step` с первым параметром, равным *n* – 1. Таким образом, в методе реализуется *рекурсивный алгоритм*. Цепочку рекурсивных вызовов метода `Step` следует прервать, когда параметр *n* станет равным `0`. Кроме того, в методе `Step` предусмотрен дополнительный вариант выхода: в случае, если была повторно нажата кнопка **Демо-режим** (для проверки этого действия анализируется свойство `Enabled` компонента `button1`). Действие 2 реализовано с помощью метода `label_Move`. После вызова этого метода необходимо вызвать метод `DoEvents` класса `Application`, который обеспечит перерисовку перемещенной метки на новом месте, а также позволит обработать нажатие кнопки **Демо-режим**, если пользователь захочет

досрочно выйти из демонстрационного режима. Наконец, вызывается метод `Sleep` класса `Thread`, приостанавливающий выполнение программы на указанный промежуток времени (этот промежуток зависит от количества блоков). Методы `DoEvents` и `Sleep` мы использовали ранее в *главе 30* (см. комментарии 2 и 3 в *разд. 30.2*).

33.7. Настройка идентификационных данных приложения

Программа HTOWERS, разработанная в данной главе, несмотря на ее небольшой размер, является законченным и достаточно интересным приложением. Поэтому имеет смысл снабдить эту программу набором стандартных характеристик, называемых *идентификационными данными* приложения. Для ввода этих данных выполните команду меню **Project | HTOWERS Properties...**, в появившейся вкладке **HTOWERS** редактора кода выберите раздел **Application** и нажмите кнопку **Assembly Information...**. На экране будет отображено окно **Assembly Information**, позволяющее задать различные идентификационные данные проекта. Введите данные в соответствии с листингом 33.14 (в результате окно **Assembly Information** примет вид, приведенный на рис. 33.5), после чего нажмите кнопку **OK** или клавишу <Enter> и повторно откомпилируйте приложение, нажав клавишу <F5>.

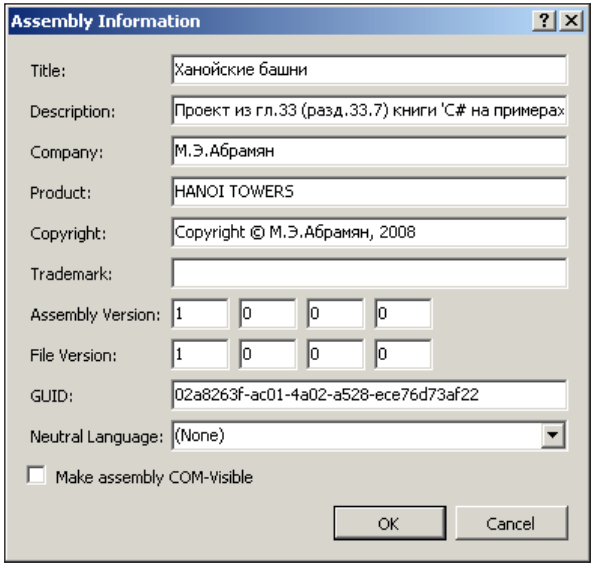


Рис. 33.5. Вид окна **Assembly Information** после ввода идентификационных данных для приложения HTOWERS

Листинг 33.14. Настройка свойств в окне *Assembly Information*

```
Title: Ханойские башни
Description: Проект из гл. 33 (разд. 33.7) книги 'C# на примерах'
Company: М. Э. Абрамян
Product: HANOI TOWERS
Copyright: Copyright © М. Э. Абрамян, 2008
```

ПРИМЕЧАНИЕ

В разделе **Application** можно также указать *значок*, который будет связан с exe-файлом приложения (см. комментарий 2 в *разд. 7.4*).

Результат. Введенные идентификационные данные сохранены в exe-файле *(сборке)* HTOWERS.exe, расположенном в подкаталоге \bin\Debug каталога проекта HTOWERS. Просмотреть эти данные можно на вкладке **Версия** окна свойств exe-файла. Напомним, что в приложении Мой компьютер или Проводник для вызова окна свойств файла достаточно выполнить команду **Свойства** из его контекстного меню. Кроме того, такие характеристики exe-файла, как его описание, производитель и версия, отображаются в виде всплывающей подсказки при наведении курсора мыши на файл в окне приложений Мой компьютер или Проводник.

Комментарии

1. Идентификационные данные проекта можно определить, и не используя окно **Assembly Information**. Достаточно загрузить в редактор файл AssemblyInfo.cs, автоматически создаваемый для любого проекта, и откорректировать его текст. Для загрузки файла AssemblyInfo.cs проще всего развернуть раздел **Properties** в окне **Solution Explorer** и выполнить двойной щелчок мышью на имени этого файла. Если загрузить файл AssemblyInfo.cs после выполнения всех действий, описанных ранее, то в нем можно будет увидеть все введенные данные (листинг 33.15).

Листинг 33.15. Фрагмент файла AssemblyInfo.cs с идентификационными данными сборки

```
[assembly: AssemblyTitle("Ханойские башни")]
[assembly: AssemblyDescription("Проект из гл. 33 (разд. 33.7) книги 'C#
↳ на примерах'")]
```

```
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("М.Э.Абрамян")]
[assembly: AssemblyProduct("HANOI TOWERS")]
[assembly: AssemblyCopyright("Copyright © М.Э.Абрамян, 2008")]
```

ПРИМЕЧАНИЕ

В коде используется значок переноса на новую строку ↳, который указывает, что эта строка является продолжением предыдущей, так как если рассматривать этот текст как две отдельные строки, то он будет синтаксически ошибочным. Разумеется, при наборе кода этот значок вставлять не надо.

2. В файле AssemblyInfo.cs можно откорректировать номер текущей версии проекта, указав его в следующих двух операторах:

```
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

В обоих операторах желательно указывать один и тот же номер версии.

При использовании окна **Assembly Information** номер версии следует ввести в поля **Assembly Version** и **File Version** (см. рис. 33.5).

Номер версии состоит из четырех чисел, разделенных точками. Первые два числа определяют собственно версию (ее старший и младший номер), третье число — номер *компоновки* в пределах текущей версии, а четвертое число — номер *ревизии* в пределах текущей компоновки. Можно, например, увеличивать номер компоновки каждый день работы над текущей версией проекта, при этом номер ревизии будет определять порядковый номер компоновки в течение дня. Впрочем, для небольших проектов вполне допустимо ограничиться указанием собственно номера версии, оставив для номеров компоновки и ревизии нулевые значения.

Вопросы, связанные с заданием номера версии и других идентификационных данных приложения, подробно рассматриваются в главах 2 книг [7] и [8].

ПРИЛОЖЕНИЯ

Приложение 1

Краткий словарь используемых терминов

После названия термина указывается его английский эквивалент. Перекрестные ссылки выделяются *курсивом*.

null

Ключевое слово языка C#, обозначающее пустую ссылку. Значение `null` может быть присвоено любой переменной классового типа для того, чтобы показать, что данная переменная не связана ни с каким реальным *объектом*.

this

Ключевое слово языка C#, доступное для использования в любом *методе класса* и обозначающее *объект*, вызвавший этот метод. Поскольку все упоминаемые в методе *члены* класса по умолчанию связываются с тем объектом, который вызвал данный метод, слово `this` явно указывается обычно лишь при передаче текущего объекта в другой метод в качестве параметра. Примером может служить использование слова `this` в качестве параметра `sender` при явном вызове обработчиков *событий* (см., например, *разд. 5.4*). В редакторе среды Visual C# слово `this` часто используется для того, чтобы не набирать вручную имена членов класса, так как список всех членов класса автоматически появляется на экране после ввода текста `this.` (с завершающей точкой).

Атрибут (attribute)

Класс, порожденный от класса `Attribute`, а также *объект* этого класса. Атрибуты предназначены для связывания информации, содержащейся в них, с различными элементами программного кода (классами, *структурами*, *перечислениями*, отдельными *членами* классов и т. д.), а также с *приложением* (сборкой) в целом. В языке C# атрибуты заключаются в квадратные скобки и указываются непосредственно перед тем элементом программы, дополнительную информацию о котором они содержат. Примером атрибута является атрибут `DllImport` из *пространства имен* `System.Runtime.InteropServices` *(см. разд. 29.1)*.

Следует отметить, что все классы-атрибуты обычно имеют имена, оканчивающиеся словом Attribute (например, `DllImportAttribute`), однако эту часть имени при их использовании в программе на языке C# можно опускать.

Делегат (delegate)

Класс, порожденный от класса `MulticastDelegate`, а также *объект* этого класса. Объект-делегат может содержать одну или несколько ссылок на *методы* и имеет средства для вызова тех методов, ссылки на которые он содержит. Делегаты применяются при определении *событий* класса, а также при передаче функций в качестве параметров (*см. разд. 32.5* и *32.6*, в которых используются параметры-функции, являющиеся делегатами-*обобщениями*).

Индексатор (indexer)

Особый вид *свойства класса*, требующий при своем вызове указания одного или нескольких параметров. В языке C# свойства-индексаторы не имеют имен; для обращения к такому свойству следует сразу после имени *объекта* указать параметры индексатора, заключив их в квадратные скобки, например: `a[i]`. Таким образом, свойства-индексаторы могут рассматриваться как аналог массивов, однако тип параметров для индексатора может отличаться от типа `int` (в частности, возможны индексаторы с параметром строкового типа `string`). Индексаторы обычно используются в классах-коллекциях. Примером *компонента*, имеющего индексаторы, является компонент-таблица `DataGridView` *(см. разд. 28.2)*.

Исключение (исключительная ситуация, exception)

Особый *класс*, а также *объект* этого класса, используемый при обработке ошибок, возникающих при выполнении *приложения*. С каждым видом ошибки связывается свое исключение. Общим предком исключений является класс `Exception`. Имена классов-исключений библиотеки классов .NET завершаются словом Exception, например, `ArgumentOutOfRangeException`.

По умолчанию любое исключение обрабатывается путем вывода на экран сообщения о возникновении исключения (после этого при выполнении программы в режиме с включенным отладчиком на экране отображается оператор, который вызвал данное исключение). Для явной обработки исключений в программе надо использовать конструкции `try-catch-finally`, называемые защищенными блоками, или `try`-блоками. Кратко опишем назначение каждого раздела `try`-блока *(см. также главу 3)*: раздел `try` содержит операторы, выполнение которых может привести к возбуждению исключения; раздел `catch(`*<имя класса-исключения>*) содержит операторы, обрабатывающие исключения ука-

занного класса (а также его классов-потомков); раздел `finally` содержит операторы, которые обязательно выполняются при выходе из `try`-блока, даже если в нем возникло исключение, не обработанное в предыдущих разделах `catch` *(см. разд. 13.6 и 13.7)*.

Допускаются сокращенные формы защищенных блоков: `try-catch` или `try-finally`. При возбуждении исключения в разделе `try` происходит немедленный переход в раздел `catch` или, если подходящих разделов `catch` нет, в раздел `finally`.

`Try`-блоки могут быть вложенными. В разделах `catch` внешнего `try`-блока обрабатываются исключения, для которых не предусмотрено обработчика во внутреннем `try`-блоке. Если ни в одном `try`-блоке не предусмотрена обработка исключения данного типа, то для него выполняется обработка по умолчанию.

Для возбуждения исключения используется оператор `throw` *(см. разд. 3.3 и 30.1)*. С помощью этого оператора можно также повторно возбудить в разделе `catch` уже обработанное исключение; это приведет к выполнению обработчиков данного исключения из внешних `try`-блоков, а при их отсутствии — к обработке исключения по умолчанию.

Класс (классовый тип, class)

Составной тип, состоящий из фиксированного числа *членов — полей*, *методов*, *свойств* и *событий*. В языке C# описывается с помощью ключевого слова `class`. При описании допускает *наследование* от уже определенных классов. Имена классов рекомендуется начинать с заглавной буквы.

Компонент (component)

Класс, порожденный от класса `Component`, а также *объект* этого класса. Почти все компоненты, за исключением тех, которые встроены в другие компоненты (например, `TabPage`, `ToolStripMenuItem` или `ToolStripButton`), располагаются в *окне компонентов* **Toolbox** и могут быть добавлены в *приложение* в режиме дизайна. *Форма* также считается особым компонентом. Многие свойства компонентов могут быть настроены в *окне свойств* **Properties**. Среди компонентов выделяют визуальные компоненты (потомки класса `Control`), которые отображаются в форме при выполнении программы.

Конструктор (constructor)

Особый *метод класса*, обеспечивающий инициализацию *объектов* этого класса. Вызывается следующим образом:

<имя инициализируемого объекта> = new *<имя класса>*(*<список параметров>*);

Попытка доступа к *членам* неинициализированного объекта приводит к *исключению* `NullReferenceException`. Инициализация главной *формы* графического *приложения* выполняется в методе `Main`, который описывается в файле Program.cs проекта. *Компоненты*, размещаемые в форме в режиме дизайна, инициализируются автоматически при инициализации формы.

Метод (method)

Функция, включенная в описание *класса*. Совокупность методов определяет действия, которые могут выполнять *объекты* данного класса.

Наследование (inheritance)

Механизм, позволяющий определять новые *классы* на основе уже существующих. При этом новый класс ("потомок") автоматически получает все *члены* существующего класса ("непосредственного предка"). *Методы* и *свойства* предка в классе-потомке могут быть изменены. Кроме того, в классе-потомке могут быть описаны новые члены. Все классы в .NET наследуются от базового класса `object`, передающего своим потомкам, в частности, методы `ToString` и `GetType` (метод `ToString` возвращает текстовое представление объекта, а метод `GetType` — фактический тип объекта, т. е. его *тип времени выполнения*). Все *компоненты* наследуются от класса `Component`, а все *исключения* — от класса `Exception`.

Объект (object)

Переменная *классового типа*. Перед использованием объекта необходимо его инициализировать с помощью *конструктора*. В .NET реализована ссылочная объектная модель, поэтому имя объекта фактически представляет собой ссылку на ту область памяти, которая содержит данный объект. Для доступа к *членам* объекта используется точечная нотация:

<имя объекта>.<имя члена>

Обобщенный класс (обобщение, generic)

Класс, при создании *объекта* которого требуется указать тип данных (один или несколько), с которыми будет работать создаваемый объект. Характерный пример классов-обобщений — обобщенные коллекции, определенные в *пространстве имен* `System.Collections.Generic`, в частности, обобщенные списки `List<T>`, позволяющие работать с элементами указанного типа `T`. Обобщенные списки используются в *разд. 7.1* (см. комментарий 2 в этом разделе), а также в *главах 31* и *32*. Классы-обобщения появились в .NET 2.0.

Окно компонентов (Toolbox window)

Элемент интегрированной среды Visual C#, позволяющий добавлять в *приложение компоненты* в режиме дизайна (визуальные компоненты при этом размещаются в *форме*, а невизуальные — в области невизуальных компонентов под изображением формы).

Окно свойств (Properties window)

Элемент интегрированной среды Visual C#, позволяющий настраивать *свойства* и *события форм* и *компонентов* в режиме дизайна.

Операция as

Обеспечивает приведение *объекта* к новому *классовому типу*. Используется в выражении

```
<имя объекта> as <имя класса>
```

которое позволяет обращаться к указанному объекту как к объекту указанного класса. Если фактический тип объекта *(тип времени выполнения)* не может быть приведен к указанному классу, то возвращается значение `null`. Приведение типа часто применяется к параметру `sender` в обработчиках *событий*. Для приведения типа можно также использовать операцию вида `(`*`тип`*`)`*`объект`*, однако в этом случае при невозможности выполнить требуемое приведение возбуждается *исключение* `InvalidCastException`. См. также комментарий 2 в *разд 27.2*.

Операция is

Позволяет определить *тип времени выполнения* для данного *объекта*. Выражение

```
<имя объекта> is <имя класса>
```

возвращает `true`, если тип объекта совпадает с указанным *классом* или *наследуется* от указанного класса. В противном случае возвращается `false`. В частности, операция `is` возвращает `false`, если объект имеет значение `null`.

Перечисление (enumeration)

Тип-*структура*, содержащий набор символьных имен, каждое из которых имеет определенное целочисленное значение. В виде перечислений обычно оформляются наборы констант, связанные с другими *классами* .NET. В качестве примеров перечислений можно указать `FontStyle`, `Keys`, `KnownColor`, `MouseButtons` и др. Типы-перечисления не имеют ни *методов*, ни *свойств*, ни *событий*. Для описания перечисления в C# используется ключевое слово `enum`.

Поле (field)

Переменная, включенная в описание *класса*. Для различных *объектов* класса поля могут принимать различные значения, определяя тем самым текущее состояние объектов. Из внешних программ поля объектов, как правило, не вызываются. Поля используются при реализации *методов*, *свойств* и *событий* класса.

Приложение (application)

Синоним исполняемой программы. Различают графические приложения (GUI-приложения), использующие графический оконный интерфейс (Graphical User Interface — графический интерфейс пользователя), и консольные приложения, выполняемые в особом консольном окне (консольное окно имитирует дисплей, находящийся в текстовом режиме). Специальным видом графического приложения является MDI-приложение (Multiple Document Interface — многодокументный интерфейс), в котором подчиненные *формы* (дочерние окна) располагаются внутри главной формы (родительского окна). Исполняемые приложения являются одним из видов сборок .NET (другим видом сборок являются библиотеки, содержащиеся в dll-файлах).

Пространство имен (namespace)

Логическая сущность, позволяющая группировать родственные *классы* и обеспечивающая уникальность их имен. Имя пространства имен присоединяется к собственно имени класса, образуя полное имя класса (например, полное имя класса `Path` имеет вид `System.IO.Path`, поскольку класс `Path` определен в пространстве имен `System.IO`). Для возможности использования в программе на C# имен классов без предваряющих их пространств имен необходимо в начале программы поместить директивы `uses` с именами требуемых пространств имен, например,

```
uses System;
uses System.IO;
```

Для определения пространства имен в языке C# используется ключевое слово `namespace`.

Свойство (property)

Член класса, снабженный особыми *методами* для своего чтения и записи. Обращение к свойству из программы выглядит как обращение к *полю*, однако фактически приводит к вызову связанных с ним методов. Это обеспечивает, в частности, выполнение необходимых побочных действий при изменении свой-

ства (например, при изменении свойства `Width` для некоторого визуального *компонента* автоматически производится перерисовка этого компонента). Многие свойства компонентов отображаются в *окне свойств* **Properties** и могут изменяться на этапе проектирования. Доступ к другим свойствам возможен только во время выполнения программы. Среди этих последних свойств могут быть свойства только для чтения, непосредственное изменение которых в программе запрещено. Примеры описания свойств приводятся в *разд. 31.1, 32.1* и *32.3*.

Событие (event)

Член класса, позволяющий связывать с *объектом* данного класса обработчик события — *метод*, который вызывается в определенной ситуации (например, при получении визуальным компонентом фокуса или щелчке на нем мышью). События компонентов и связанные с ними обработчики указываются в разделе **Events** *окна свойств* **Properties**. Дополнительные сведения о подключении обработчиков к событиям, а также их последующем отключении, приводятся в *главе 4*. При реализации событий используются *делегаты*.

Статические члены (static members)

Члены класса, связанные не с конкретным *объектом* (т. е. экземпляром) класса, а с классом в целом. Статические члены называют также классовыми, в противоположность "обычным", экземплярным членам. Для вызова статических членов перед их именами надо указать имя класса, например, `Control.ModifierKeys` (статическое свойство класса `Control`) или `File.Exists("a.txt")` (статический метод класса `File`). В языке C# при описании статических членов указывается ключевое слово `static`. Для использования статических членов класса не требуется создавать объекты данного класса. Если класс содержит только статические члены (как, например, класс `File` или класс `Math`), то необходимости в создании его объектов никогда не возникает.

Структура (structure)

Так в C# и некоторых других языках платформы .NET называются особые *классы*, экземпляры которых (в отличие от экземпляров "обычных" классов) размещаются не в динамической памяти, а в стеке. В переменной структурного типа хранится не ссылка на некоторый объект, а сам объект-структура, поэтому типы-структуры называются также типами-значениями (value types). Для описания структуры в языке C# используется ключевое слово `struct` (в отличие от слова `class`, используемого при описании "обычного" класса ссылочного типа). В качестве примеров структур можно привести все числовые

типы, символьный тип `char`, типы `Color`, `Point`, `Size`, `Rectangle`, `DateTime`. В виде структур обычно описываются типы данных небольшого размера, которые неэффективно размещать в динамической памяти. Поскольку переменные структурных типов не являются ссылками, им нельзя присваивать значение `null`. Частным случаем структур являются *перечисления*.

Тип времени выполнения (run-time type)

Фактический тип *объекта*, который указывается при его инициализации с помощью *конструктора*. Может не совпадать с типом, использованным при описании данного объекта. Требуется лишь, чтобы *класс*, использованный при описании объекта, содержался в цепочке предков для класса, использованного при инициализации объекта. Например, объект, описанный как `object`, можно инициализировать как `Label` (с помощью конструктора `new Label()`), но не наоборот. Определить тип времени выполнения инициализированного объекта можно с помощью метода `GetType` или *операции is*. Метод `GetType`, в отличие от операции `as`, используется, если требуется определить точное совпадение типа времени выполнения с некоторым классом. Например, выражение `a.GetType() == typeof(Label)` вернет значение `true` только в случае, если объект `a` имеет тип `Label`; если же объект `a` имеет тип, *унаследованный* от класса `Label` (например, является меткой-ссылкой типа `LinkLabel`), то указанное выражение вернет значение `false`, тогда как выражение `a is Label` вернет `true`.

Форма (form)

Класс, *унаследованный* от класса `Form`, а также *объект* этого класса. Форма представляет собой объектную реализацию окна операционной системы Windows. Графическое *приложение* может включать несколько форм, одна из которых считается главной (главная форма отображается на экране при запуске приложения; закрытие главной формы приводит к завершению работы приложения). Каждая форма описывается в виде нового класса (потомка `Form`) в отдельном наборе файлов. *Свойства* и *события* формы, как и обычных *компонентов*, можно задать с помощью *окна свойств*.

Член (member)

Общее название для *полей*, *методов*, *свойств* и *событий класса* или *объекта*.

Приложение 2

Описание компакт-диска

Компакт-диск содержит все проекты, описанные в книге, причем для каждого проекта приводится несколько вариантов, соответствующих различным этапам его разработки (табл. П2.1). С каждым проектом связан каталог первого уровня; название каталога состоит из номера главы, в которой описывается проект, и имени проекта (например, 02-DISKINFO). Каталоги второго уровня соответствуют различным этапам разработки проекта; их название совпадает с номером раздела в пределах главы (например, 1). Каждый каталог второго уровня содержит подкаталог bin\Debug, в котором находится готовый к запуску исполняемый файл, соответствующий текущему варианту проекта. В некоторых случаях подкаталог bin\Debug включает вспомогательные текстовые или графические файлы, которые можно использовать для проверки работы исполняемого файла.

Все проекты разработаны в среде Visual C# 2005, однако могут использоваться и в среде Visual C# 2008. При загрузке проекта в среду Visual C# 2008 будет автоматически запущен мастер по преобразованию проекта (Conversion Wizard), позволяющий откорректировать настройки проекта в соответствии с требованиями новой версии интегрированной среды. Для выполнения требуемой корректировки достаточно сразу нажать кнопку **Finish** в окне **Visual Studio Conversion Wizard**, а после появления сообщения об успешном преобразовании проекта — закрыть это окно, нажав кнопку **Close**. Следует заметить, что при подобном преобразовании изменяются лишь файлы со служебной информацией (в частности, файлы с расширениями sln и csproj); файлы с текстами программ (cs-файлы) остаются неизменными.

Таблица П2.1. *Описание компакт-диска*

Каталог первого уровня	Каталоги второго уровня	Содержимое
02-DISKINFO	1, 2, 3	Варианты проекта DISKINFO, описанные в *разд. 2.1—2.3*

Таблица П2.1 *(продолжение)*

Каталог первого уровня	Каталоги второго уровня	Содержимое
03-EXCEP	1, 2, 3	Варианты проекта EXCEP, описанные в *разд. 3.1—3.3*
04-EVENTS	1, 2, 3	Варианты проекта EVENTS, описанные в *разд. 4.1—4.3*
05-FORMS	1, 2, 3, 4, 5, 6	Варианты проекта FORMS, описанные в *разд. 5.1—5.6*
06-CALC	1, 2, 3, 4, 5	Варианты проекта CALC, описанные в *разд. 6.1—6.5*
07-CURSORS	1, 2, 3, 4, 5	Варианты проекта CURSORS, описанные в *разд. 7.1—7.5*
08-TEXTBOXES	1, 2, 3, 4, 5, 6	Варианты проекта TEXTBOXES, описанные в *разд. 8.1—8.6*
09-COLORS	1, 2, 3, 4, 5, 6	Варианты проекта COLORS, описанные в *разд. 9.1—9.6*
10-MOUSE	1, 2, 3, 4, 5, 6, 7	Варианты проекта MOUSE, описанные в *разд. 10.1—10.7*
11-ZOO	1, 2, 3, 4, 5, 6, 7, 8	Варианты проекта ZOO, описанные в *разд. 11.1—11.8*
12-CLOCK	1, 2, 3, 4	Варианты проекта CLOCK, описанные в *разд. 12.1—12.4*
13-IMGVIEW	1, 2, 3, 4, 5, 6, 7	Варианты проекта IMGVIEW, описанные в *разд. 13.1—13.7*
14-PNGEDIT1	1, 2, 3, 4	Варианты проекта PNGEDIT1, описанные в *разд. 14.1—14.4*
15-PNGEDIT2	1, 2	Варианты проекта PNGEDIT2, описанные в *разд. 15.1—15.2*
16-PNGEDIT3	1, 2, 3, 4	Варианты проекта PNGEDIT3, описанные в *разд. 16.1—16.4*
17-PNGEDIT4	1, 2, 3, 4, 5	Варианты проекта PNGEDIT4, описанные в *разд. 17.1—17.5*
18-TXTEDIT1	1, 2, 3, 4	Варианты проекта TXTEDIT1, описанные в *разд. 18.1—18.4*
19-TXTEDIT2	1, 2, 3, 4	Варианты проекта TXTEDIT2, описанные в *разд. 19.1—19.4*

Таблица П2.1 *(окончание)*

Каталог первого уровня	Каталоги второго уровня	Содержимое
20-TXTEDIT3	1, 2, 3	Варианты проекта TXTEDIT3, описанные в *разд. 20.1—20.3*
21-TXTEDIT4	1, 2	Варианты проекта TXTEDIT4, описанные в *разд. 21.1—21.2*
22-TXTEDIT5	1, 2, 3, 4	Варианты проекта TXTEDIT5, описанные в *разд. 22.1—22.4*
23-TXTEDIT6	1, 2, 3, 4, 5	Варианты проекта TXTEDIT6, описанные в *разд. 23.1—23.5*
24-CHKBOXES	1, 2, 3	Варианты проекта CHKBOXES, описанные в *разд. 24.1—24.3*
25-LISTBOX1	1, 2, 3, 4	Варианты проекта LISTBOX1, описанные в *разд. 25.1—25.4*
26-LISTBOX2	1, 2, 3	Варианты проекта LISTBOX2, описанные в *разд. 26.1—26.3*
27-JPEGVIEW	1, 2, 3, 4, 5, 6, 7, 8	Варианты проекта JPEGVIEW, описанные в *разд. 27.1—27.8*
28-FONTVIEW	1, 2, 3, 4	Варианты проекта FONTVIEW, описанные в *разд. 28.1—28.4*
29-ICONVIEW	1, 2, 3, 4, 5, 6	Варианты проекта ICONVIEW, описанные в *разд. 29.1—29.6*
30-TRIGFUNC	1, 2, 3, 4, 5, 6	Варианты проекта TRIGFUNC, описанные в *разд. 30.1—30.6*
31-STUD1	1, 2, 3, 4, 5, 6, 7	Варианты проекта STUD1, описанные в *разд. 31.1—31.7*
32-STUD2	1, 2, 3, 4, 5, 6	Варианты проекта STUD2, описанные в *разд. 32.1—32.6*
33-HTOWERS	1, 2, 3, 4, 5, 6, 7	Варианты проекта HTOWERS, описанные в *разд. 33.1—33.7*

Литература

1. Балена Ф., Димауро Дж. Современная практика программирования на Microsoft Visual Basic и Visual C#. — М.: Русская редакция, 2006.
2. Грэхэм Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики. — М.: Мир, 1998.
3. Нортроп Т., Уилдермьюс Ш., Райан Б. Основы разработки приложений на платформе Microsoft .NET Framework. Учебный курс. — М.: Русская редакция, СПб.: Питер, 2007.
4. Петцольд Ч. Программирование в тональности C#. — М.: Русская редакция, 2004.
5. Петцольд Ч. Программирование для Microsoft Windows на C# (в 2-х т.). — М.: Русская редакция, 2002.
6. Петцольд Ч. Программирование с использованием Microsoft Windows Forms. — М.: Русская редакция, СПб.: Питер, 2006.
7. Рихтер Дж. Программирование на платформе Microsoft .NET Framework. — М.: Русская редакция, СПб.: Питер, 2005.
8. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. — М.: Русская редакция, СПб.: Питер, 2007.

Предметный указатель