

Программирование для Microsoft® WINDOWS на C#

Charles Petzold

Programming Microsoft® WINDOWS with C#

Microsoft Press

Чарльз Петцольд

Программирование для Microsoft® WINDOWS на C#

Том 2

Москва 2002

РУССКАЯ РЕДАКЦИЯ

УДК 004.43
ББК 32.973.26-018.2
П33

Петцольд Ч.

П33 Программирование для Microsoft Windows на C#. В 2-х томах. Том 2./ Пер. с англ. — М,: Издательско-торговый дом «Русская Редакция», 2002. — 624 с.: ил.

ISBN 5-7502-0220-8

Ч. Петцольд, известный автор и один из пионеров Windows-программирования, в этой книге подробно и доходчиво рассказывает о возможностях Windows Forms — библиотеки классов Windows нового поколения для платформы .NET. Вы узнаете, как создавать динамические пользовательские интерфейсы, реализовывать графический вывод, управлять клавиатурой, мышью и таймером на C#. Подробное описание языковых конструкций, сравнение их с другими популярными языками программирования и примеры программ помогут вам быстро освоить этот новый объектно-ориентированный язык. В книге подробно рассмотрена иерархия классов .NET Framework, благодаря чему вы сможете приступить к разработке собственных программ с применением Windows Forms.

Том 2 состоит из 13 глав и предметного указателя.

УДК 004.43
ББК 32.973.26-018

Подготовлено к изданию по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон. США.

Macintosh — охраняемый товарный знак компании Apple Computer Inc. ActiveX, BackOffice, JScript. Microsoft. Microsoft Press. MSDN. NetShow, Outlook, PowerPoint, Visual Basic, Visual C++, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, Win32, Windows и Windows NT являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах. Все другие товарные знаки являются собственностью соответствующих фирм

Все названия компаний, организаций и продуктов, а также имена лиц, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам и лицам.

© Оригинальное издание на английском языке, Charles Petzold, 2002

© Перевод на русский язык, Microsoft Corporation, 2002

© Оформление и подготовка к изданию, издательско-торговый дом «Русская Редакция», 2002

ISBN 0-7356-1370-2 (англ.)
ISBN 5-7502-0220-8

Содержание

I ТОМ

Общее

1 Работаем с консолью
2 Hello, Windows Forms
3 Основные структуры
4 Упражнение по выводу текста

Графика

5 Линии, кривые и заливка областей
7 Страницы и преобразования
9 Текст и шрифты

Пользовательский интерфейс

6 Работа с клавиатурой
8 Приручение мыши
10 Таймер и время
11 Изображения и битовые карты

Приложения

А Файлы и потоки
Б Математические функции
В Работа со строками

II ТОМ

Графика

13 Кривые Безье и другие сплайны
15 Контуры, области и отсечение
17 Кисти и перья
19 Игры со шрифтами
21 Печать
23 Метафайлы

Пользовательский интерфейс

12 Кнопки, надписи и полосы прокрутки
14 Меню
16 Диалоговые окна
18 Текстовые поля, списки и управляющие стрелки
20 Панели инструментов и строки состояния
22 Деревья и списки
24 Буфер обмена и буксировка

Оглавление II тома

Глава 12 Кнопки, надписи и полосы прокрутки 549

Кнопки и щелчки ... 550
Клавиатура и мышь ... 553
Вопросы, связанные с элементами управления ... 554
Подробнее о кнопках ... 555
Внешний вид и выравнивание ... 558
Кнопки с изображениями ... 561
Несколько обработчиков или один? ... 564
Прорисовка владельцем ... 564
Бросаем якорь ... 569
Производим стыковку ... 572
Дочерние элементы формы ... 575
Z-порядок ... ?!77
Флажок ... 578
Вариант с тремя состояниями ... 581
Элемент управления «надпись» ... 582
Порядок перехода и остановки по клавише Tab ... 585
Идентификация элементов управления ... 585
Автоматическое масштабирование ... 588
Автоматическое масштабирование и Windows Forms Designer ... 590
Творческий подход к AutoScaleBaseSize ... 591
Внутри автоматического масштабирования ... 591
Шестнадцатеричный калькулятор ... 593
Переключатели и групповые блоки ... 598
Полосы прокрутки ... 601
Регулятор как альтернатива полосам прокрутки ... 609

Глава 13 Кривые Безье и другие сплайны 614

Знакомство с кривыми Безье на практике ... 615
Стильные часы ... 619
Коллинеарные точки и кривые Безье ... 622
Рисование окружностей и дут кривыми Безье ... 624
«Картины Безье» ... 627
Вывод формул Безье ... 628
Канонический сплайн ... 632
Вывод уравнений канонической кривой ... 638

Глава 14 Меню 642

Меню и его пункты ... 642
Быстрые клавиши меню ... 645
Ваше первое меню ... 647
Необычные меню ... 650
Свойства и события MenuItem ... <>52

Отмеченные пункты меню ... 654
Работа с контекстными меню ... 658
Набор пунктов меню ... 661
Стандартные меню (предложение) ... 667
Прорисовка владельцем ... 672

Глава 15 Контуры, области и отсечение 681

Задача и ее решение ... 681
Более формально о контурах ... 686
Создание контура ... 688
Визуализация контура ... 692
Преобразования контуров ... 696
Другие виды модификации контура ... 698
Отсечение при помощи контуров ... 705
Отсечение растровых изображений ... 710
Области и отсечение ... 714

Глава 16 Диалоговые окна 716

Ваше первое модальное диалоговое окно ... 717
Закрытие модального диалогового окна ... 721
Accept и Cancel ... 723
Размещение на экране ... 725
Окно About ... 728
Определение свойств диалоговых окон ... 731
Реализация кнопки Apply ... 736
Немодальные диалоговые окна ... 740
Общие диалоговые окна ... 743
Выбор шрифта и цвета ... 744
Использование реестра Windows ... 751
Диалоговое окно Open File ... 755
Диалоговое окно Save File ... 763

Глава 17 Кисти и перья 766

Заливка сплошными цветами ... 767
Узорные кисти ... 767
Начальная точка визуализации узора ... 776
Текстурные кисти ... 779
Кисти с линейным градиентом ... 783
Кисти с множественным градиентом ... 791
Мозаичное расположение кисти ... 796
Перья тоже могут быть кистями ... 804
Оформление пунктирных линий ... 807
Концы и соединения линий ... 810

Глава 18 Текстовые поля, списки и управляющие стрелки 817

Однострочные текстовые поля ... 817
Многострочные текстовые поля ... 822
Клонирование Notepad ... 824

Клон Notepad с файловым вводом-выводом ... 829
Продолжаем разрабатывать клон Notepad ... 839
Использование текстовых полей в специальных целях ... 854
Поле ввода с форматированием ... 855
Подсказки ... 856
Окно списка ... 863
Окно списка + Текстовое поле = Комбинированное окно ... 869
Управляющие стрелки ... 875

Глава 19 Игры со шрифтами 887

Приступим к делу ... 887
Закрашивание текста кистью ... 889
Преобразование шрифта ... 396
Текст и контуры ... 906
Нелинейные преобразования ... 920

Глава 20 Панели инструментов и строки состояния 927

Простая строка состояния ... 928
Строка состояния и автоматическая прокрутка ... 930
Панели строки состояния ... 933
Свойства класса StatusBarPanel ... 935
Подсказки для меню ... 939
Простая панель инструментов ... 945
Разновидности панелей инструментов ... 949
События панели инструментов ... 952
Стили панели инструментов ... 957

Глава 21 Печать 966

Принтеры и их параметры ... 966
Параметры страницы ... 973
Определение документа для печати ... 976
Обработка событий объекта PrintDocument ... 978
Размеры страницы ... 984
Класс PrintController ... 988
Стандартное диалоговое окно печати ... 992
Настройка страницы ... 996
Предварительный просмотр ... 1000

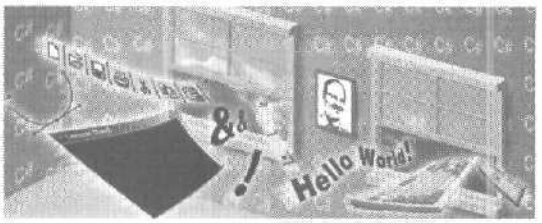
Глава 22 Деревья и списки 1008

В мире разделителей ... 1008
Деревья и их узлы ... 1023
Изображения в деревьях ... 1026
События дерева ... 1027
Перемещение по узлам ... 1029
Дерево каталогов ... 1031
Показ изображений ... 1036
Основные сведения о списках ... 1044
События списков ... 1050

Глава 23 Метафайлы 1058
Загрузка и отображение метафайлов 1059
Размеры и параметры отображения метафайлов 1061
Преобразование метафайлов в растровые изображения 1068
Создание новых метафайлов 1071
Ограничивающий прямоугольник 1078
Метафайлы и преобразования страниц 1080
Типы метафайлов 1083
Обработка записей метафайла 1085
Глава 24 Буфер обмена и буксировка 1093
Элементы и форматы 1094
Очень маленький (но мощный) класс Clipboard 1094
Получение объектов из буфера обмена 1096
Форматы данных буфера обмена 1105
Программы просмотра буфера обмена 1114
Использование нескольких форматов буфера обмена 1124
Буксировка 1128
Предметный указатель 1136

ГЛАВА

12



Кнопки, надписи и полосы прокрутки

Одно из важнейших преимуществ ОС Microsoft Windows, как и других графических интерфейсов, — возможность задействовать хорошо знакомые графические объекты. Полосы прокрутки, меню, кнопки, переключатели, флажки, текстовые поля ввода, списки — все это *элементы управления.* В графическом интерфейсе пользователя элементы управления — это то же, что подпрограммы в языках программирования. Элементы управления позволяют создавать структурированные и модульные приложения, не отвлекаясь на низкоуровневую работу с клавиатурой и мышью.

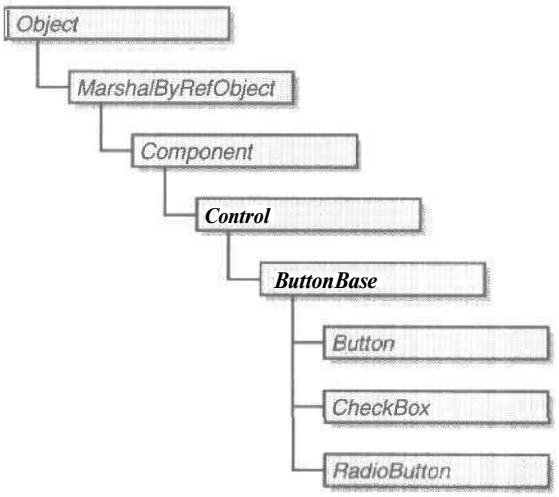
На заре Windows элементы управления называли *дочерними окнами.* Кроме меню и полос прокрутки, они обычно встречались в диалоговых окнах. Элементы управления можно было помещать в главное окно приложения, но это случалось редко, и обычно так никто не делал. Так было до 1991 г. — до новой парадигмы программирования для Windows, реализованной в Microsoft Visual Basic. Visual Basic позволил программистам интерактивно размещать элементы управления в главном окне приложения, а затем писать связанный с этими элементами код. Такой стиль хорошо зарекомендовал себя при написании клиентских интерфейсов распределенных приложений. Кроме того, в этом стиле написаны многие прил' жения, в которых главное — интерфейс пользователя (например, Windows CD Player).

Я уже приводил примеры элементов управления: в главе 4 я использовал элемент управления *Panel,* в главе 8 — *Label,* а в главах 8 и 10 — пользовательские элементы управления, считавшиеся довольно сложными. Пора изучить предопределенные элемента управления, доступные в .NET Framework, в системе.

Кнопки и щелчки

Наверное, основной элемент управления — кнопка, этот вездесущий прямоугольный объект, на котором часто написано OK, Отмена, Открыть или Сохранить, Кнопки (иногда их называют *командными*) предназначены для немедленного выполнения некоего действия. Кнопку можно либо щелкнуть мышью, либо, если фокус ввода находится на кнопке, нажать пробел. Иногда можно нажать кнопку, даже когда на ней не находится фокус ввода, нажав Enter или Esc. Далее в этой главе я рассмотрю использование клавиши Enter, а еще об Enter и Esc мы поговорим в главе 16.

Кнопка реализована в классе *Button,* являющимся одним из трех наследников абстрактного класса *ButtonBase*:



Классы *CheckBox* и *RadioButton* мы обсудим ниже,

В программе Windows Forms элемент управления «кнопка» создается при создании объекта типа *Button*. Кнопка сообщает форме о том, что ее нажали, используя установленный для нее обработчик события *Click*. Программа SimpleButton создает единственную кнопку, при щелчке которой форма выводит некий текст.

SimpieButton.cs

```
//---------------------------------------
// SimpleButton.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleButton: Form
{
     public static void Main()
     {
          Application.Run(new SimpleButton());
```

см. след. стр.

```
    }
    public SimpleButton()
    {
        Text = "Simple Button";

        Button btn   = new Button();
        btn.Parent   = this;
        btn.Text     = "Click Me!";
        btn.Location * new Point(100, 100);
        btn.Click   += new EventHandler(ButtonOnClick);
    I
    void ButtonOnClick(object obj, EventArgs ea)
    {
        Graphics grfx   = CreateGraphics();
        Point    ptText = Point.Empty;
        string   str    = "Button clicked!";

        grfx.DrawString(str, Foot, new SolidBrush(ForeColor), ptText);
        System.Threading, Thread.Sleep(250);
        grfx.FillRectangle(new SolidBrush(BackColor),
            new RectangleF(ptText, grfx.MeasureString(str, Font)));

        grfx.Dispose();
    }
}
```

Задав текст заголовка окна, конструктор формы создает объект типа *Button:*

```
Button btn = new Button();
```

Класс *Button* имеет только конструктор по умолчанию.

Далее следует сделать одну вещь, о которой лично я часто забываю, поэтому я и приучил себя делать это сразу после создания элемента управления: нужно указать его родительский элемент. Это позволяет сделать свойство *Parent* элемента управления:

```
btn.Parent = this;
```

Ключевое слово *this* указывает на текущий объект, в нашем случае на форму - объект типа *SimpleButton* был создан в методе *Main.* Так что кнопка становится дочерним элементом формы,

Концепция родительских отношений применяется в программировании широко. Так, существуют родительские и дочерние процессы, родительские и дочерние папки, родительские классы и классы-потомки. Следовательно, концепция родительских отношений является базовой для языков программирования и ОС. Элементы управления тоже должны иметь предков! Элемент управления без предка невидим. Когда вы задаете свойство элемента управления *Parent,* вы на самом деле указываете следующую связь между родителем и элементом управления.

- Дочерний элемент управления располагается на поверхности предка. Более того, элемент управления *привязан* к нему, т. е. он не может выходить за границы области. занимаемой предком,

- Расположение дочернего элемента указывается относительно левого верхнего угла клиентской области предка,
- Потомок наследует свойства своего предка, в частности *Font, ForeColor* и *BackColor*.

Вернемся к SimpleButton. Следующее выражение в конструкторе присваивает свойству кнопки *Text* текст:

```
btn.Text = "Click Me!";
```

который выводится на поверхности кнопки. Выражение:

```
btn.Location = new Point(100, 100);
```

указывает, где должен находиться левый верхний угол кнопки относительно левого верхнего угла клиентской области родителя. Здесь я просто прикинул, где примерно будет располагаться центр формы, и поместил кнопку туда. Далее я покажу, как определять координаты точнее. В *Control* реализованы все свойства расположения и размера, приведенные в главе 3. Для любых потомков *Control,* кроме *Form,* значение свойства *ClientSize* обычно равно значению свойства *Size.*

Кстати о размере. Вы, наверное, заметили, что я не указал размер кнопки. Я не сделал этого потому, что надеюсь, что кнопка с размером по умолчанию отлично подойдет для наших целей. А если нет, то неудачный размер кнопки будет нам хорошим уроком.

И наконец, форма устанавливает для кнопки обработчик события *Click*:

```
btn.Click += new EventHandler(ButtonOnClick);
```

Кнопка вызывает событие *Click,* когда ее щелкают или если на ней находится фокус ввода, при нажатии пробела. Так как далее мы будем устанавливать много обработчиков событий, я буду придерживаться стандартной схемы наименования. В этой схеме имя обработчика события состоит из типа объекта (у нас это *Button),* возможно, еще одного описательного слова, если класс содержит более одного обработчика для конкретного события объекта конкретного типа, слова *On* и имени события — у нас это *Click*.

Метод *ButtonOnClick* должен быть определен в соответствии с делегатом *EventHandler.* Метод принимает два параметра:

```
void ButtonOnClick(object obj, EventArgs ea)
{
    :
}
```

В SimpleButton событие *ButtonOnClick* просто выводит строку текста в левом верхнем углу клиентской области и стирает ее через 1/4 секунды.

Заметьте: хотя *ButtonOnClick* обрабатывает событие *Click* кнопки, обработчик сообщения является методом класса *SimpleButton.* Так, когда метод *ButtonOnClick* вызывает *CreateGraphics,* он создает объект *Graphics,* относящийся к форме, а не к кнопке. Если в методе *ButtonOnClick* надо обратиться к свойствам или методам кнопки, нужно привести параметр *object* примерно так:

```
Button btn = (Button) obj;
```

С другой стороны, объект *Button* может храниться как поле класса *SimpleButton*.

Кроме того, не надо путать метод, названный мной *ButtonOnClick*, с обычным методом *OnClick*. Если бы в SimpleButton был бы переопределен метод *OnClick*, этот метод получал бы сообщения о щелчках формы, а не кнопки. При щелчке кнопки вызывается метод *ButtonOnClick*. Щелчком в другом месте клиентской области вызывается метод *OnClick*.

Окно программы SimpleButton выглядит так:



Нажать кнопку (и инициировать вывод текста методом *ButtonOnClick*) можно мышью или нажатием клавиш «пробел» либо Enter,

Клавиатура и мышь

Я уже говорил, как форма *SimpleButton* и кнопка реагируют на мышь. Если курсор расположен над кнопкой, она получает событие от мыши, если в другом месте клиентской области формы — событие от мыши получает форма. (Но не забудьте о захвате мыши: при щелчке над кнопкой кнопка захватывает мышь и продолжает получать *все* события от нее, пока кнопка мыши не будет отпущена или командная кнопка не потеряет захват мыши.)

С точки зрения клавиатурного ввода, различие между SimpleButton и большинством предыдущих программ из этой книги гораздо существеннее: в любой момент, когда активно окно SimpleButton, кнопка имеет фокус ввода, т. е. форма *не* получает клавиатурного ввода,

В этом можно убедиться. Если вы переопределите в классе *SimpleControl* методы *OnKeyDown*, *OnKeyUp*, и *OnKeyPress*, станет очевидно, что форма не получает событий нажатия клавиш. Можете установить обработчики событий *KeyDown*, *KeyUp* и *KeyPress* для кнопки. Для этого в класс *SimpleButton* нужно добавить примерно такие методы:

```
void ButtonOnKeyDown(object obj, KeyEventArgs kea)
{
    ⋮
}
```

Придется установить и обработчики событий для кнопки, например так:

```
btn.KeyDown += new KeyEventHandler(ButtonOnKeyDown);
```

Установив обработчики событий от клавиатуры для кнопки, вы увидите, что, когда окно программы SimpleButton активно, *большинство* событий нажатия клавиш передается кнопке. Сама по себе кнопка игнорирует большинство этих событий, кроме нажатия пробела.

Я сказал, что кнопка получает *большинство* клавиатурных событий, О нажатии некоторых клавиш ни кнопка, ни форма не узнают никогда. Такими событиями являются событие *KeyDown* для клавиш Enter, Tab и стрелок (вверх, вниз, влево и вправо), а также события *KeyPress* для клавиш Enter и Tab.

Клавиши-стрелки и Tab обычно служат для перемещения между элементами управления в диалоговых окнах Windows, а нажатие Enter обычно заставляет сработать кнопку по умолчанию. Нажатия этих клавиш перехватывает код в классе *ContainerControl*. Вы, наверное, помните, что *ContainerControl* — один из предков *Form*. Класс *ContainerControl* отвечает за передачу фокуса ввода между дочерними элементами управления. Поэтому класс типа *Form*, предназначенный для хранения набора элементов управления, часто называют *контейнером* элементов управления.

Элемент управления получает ввод с клавиатуры и мыши, только если он видим и не заблокирован, т. е. как свойство *Visible,* так и свойство *Enabled* должны быть установлены в *true*:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
bool	*Visible*	Чтение/запись
bool	*Enabled*	Чтение/запись

Оба этих свойства по умолчанию установлены в *true*. Если установить свойство *Visible* в *false,* элемент управления исчезнет и не будет получать ввод с клавиатуры или мыши. Если элемент управления размещен на форме, ввод с мыши, который он должен был получать, получит форма, словно элемента управления там нет вовсе. Если невидимый элемент управления — единственный на форме, ввод с клавиатуры, как правило, получает форма.

Если установить свойство *Enabled* элемента управления в *false,* элемент управления будет видим, но не будет отвечать на ввод с клавиатуры или мыши. Часто надписи серого цвета на элементах управления указывают на то, что они заблокированы,

Вопросы, связанные с элементами управления

SimpleButton может показаться простой программой, но в ней я демонстрирую один весьма спорный подход. Думаю, многие, увидев такую программу, недоуменно покачают головой. Можете в это поверить? Так что же я делаю такого плохого?

Я пишу код вручную!

Возможно, вы знаете, что Visual C# .NET включает в себя Windows Forms Designer (конструктор форм Windows) — инструмент, позволяющий интерактивно проектировать форму, выбирая и размещая на ней элементы управления, а затем писать код, связанный с каждым таким элементом. Windows Forms Designer генерирует код в файле .cs, который создает эти элементы и задает их свойства. Преимущества Windows Forms Designer очевидны: вы можете размещать элементы управ-

ления, как хотите (например, в соответствии с вашими представлениями о красоте), и при этом не думать о конкретных числах, определяющих их местоположение и размер.

Но по большей части в этой книге я притворяюсь, что конструктора нет. Все свойства элементов управления я задаю вручную. У ручного кодирования размеров и местоположения элементов управления есть определенные преимущества: например, можно использовать значения *const*... Или переменные... Или массивы... Или циклы *for*... И я продемонстрирую эти возможности!

Но еще важнее то, что Windows Forms Designer поможет только до определенного момента, а потом вам придется писать код и, возможно, разбираться в том коде, что генерирует для вас Windows Forms Designer.

Поймите меня правильно: Windows Forms Designer — это очень хорошо, но... Не уметь кодировать элементы управления самостоятельно — это *плохо*.

Подробнее о кнопках

Похоже, никакой катастрофы из-за того, что я не установил свойство *Size* объекта *Button* в программе SimpleButton, не случилось. На моем компьютере (и на нашем, возможно, тоже) кнопка получилась примерно такого размера, как нужно. Но если установить свойства дисплея на использование очень большого шрифта (больше Large Fonts), часть текста на кнопке может быть обрезана. А это не очень-то хорошо,

Когда вы размещаете на форме элементы управления, первейшим критерием должно быть удобство использования. Этот критерий включает в себя массу факторов. Так, элементов управления не должно быть слишком много, они должны быть выстроены в логическом порядке. Неплохо, когда форма эстетически гармонична. Но главное, текст на элементе управления не должен обрезаться! Может, пользователь и догадается, что «Cance» значит «Cancel» (Отмена), но программиста это характеризует не с лучшей стороны.

Как я покажу чуть позже, изменять размеры элементов управления можно по-разному. Если у вас есть опыт создания независимой от устройств графики, включающей текст, он вам пригодится. Но с элементами управления связаны определенные трудности. Например, если установить высоту кнопки равной высоте шрифта, текст все равно будет обрезан из-за наличия границы. Ширина границы кнопки — 4 пиксела, но никакого источника информации на этот счет нет, а кроме того, эта величина может быть другой при экзотических разрешениях экрана

Так какой же высоты должна быть кнопка? Обычно рекомендуется 7/4 (или 175%) от высоты шрифта. В следующей программе высота кнопок вдвое больше высоты шрифта, и надо сказать, кнопки не кажутся слишком уж большими. Очень важно протестировать программу при разных разрешениях,

Если вы решите перевести приложение на другие языки, возникнут дополнительные проблемы. Перевод влияет на ширину текстовых строк: одни языки «многословнее» других.

В следующей программе две кнопки используются для демонстрации работы с размерами кнопок. Нажатие одной кнопки делает форму на 10% меньше, другой — на 10% больше (в пределах налагаемых Windows ограничений). Кнопки остаются в центре клиентской области.

TwoButtons.cs

```
//-------------------------------------------
// TwoButtons.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoButtons: Form
{
     readonly Button btnLarger, btnSmaller;
     readonly int    cxBtn, cyBtn, dxBtn;
     public static void Main()
     I
          Application.Run(new TwoButtons());
     }
     public TwoButtons()
     {
          Text = "Two Buttons";
          ResizeRedraw = true;

          cxBtn - 5 * Font.Height;
          cyBtn = 2 * Font.Height;
          dxBtn =     Font.Height;

          btnLarger = new fiutton();
          btnLarger.Parent = this;
          btnLarger.Text   = "&Larger";
          btnLarger.Size   = new Size(cxBtn, cyBtn);
          btnLarger.Click += new EventHandler(ButtonOnClick);

          btnSmaller * new Button();
          btnSmaller.Parent = this;
          btnSmaller.Text   = "&Smaller";
          btnSmaller.Size   = new Size(cxBtn, cyBtn);
          btnSmaller.Click += new EventHandler(ButtonOnClick);

          OnResize(EventArgs.Empty);
     }
     protected override void OnResize(EventArgs ea)
     {
          base.OnResize(ea);

          btnLarger.Location =
                         new Point(ClientSize.Width / 2 - cxBtn - dxBtn / 2,
                                   (ClientSize.Height - cyBtn) / 2);
          btnSmaller.Location *
                         new Point(ClientSize.Width / 2 * dxBtn / 2,
                                   (ClientSize.Height - cyBtn) / 2);
```

см. след. стр.

```
    }
    void ButtonOnClick(object obj, EventArgs ea)
    {
        Button btn = (Button) obj;

        if (btn = btnLarger)
        {
            Left   -= (int)(0.05 * Width);
            Top    -= (int)(0.05 * Height);
            Width  += (int)(0.10 * Width);
            Height += (int)(0.10 * Height);
        }
        else
        {
            left   += (int)(Width  / 22f);
            Top    += (int)(Height / 22f);
            Width  -= (int)(Width  / 11f);
            Height -= (int)(Height / 11f);
        }
    }
}
```

Конструктор вычисляет три значения и хранит их в полях: *cxBtn* и *cyBtn* определяют ширину и высоту каждой кнопки, а *dxBtn* — расстояние между двумя кнопками. Все три значения основываются на свойстве *Height* свойства *Font* формы. Так как элементы управления наследуют последнее свойство от предков, такой же размер имеет шрифт кнопок. Высота кнопок устанавливается вдвое больше высоты шрифта, а их ширина — впятеро больше ширины шрифта. (Я выбрал пять, так как для данной программы это неплохо работает, другие подходы мы обсудим потом.) Конструктор устанавливает только размер каждой кнопки, а не их расположение.

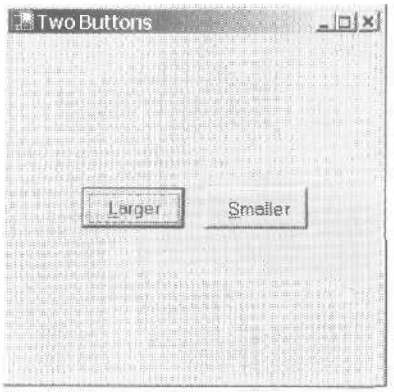
Так как расположение кнопки зависит от размера клиентской области, оно не устанавливается до тех пор. пока не вызывается метод *OnResize*. В первый раз он вызывается в последнем операторе конструктора.

Текст каждой кнопки начинается с амперсанта (&), что приводит к тому, что первая буква подчеркивается. Подчеркнутая буква работает как клавиша быстрого доступа. Во время работы программы можно на краткое время нажать клавишу Alt, и отобразятся подчеркивания (см. рис. на следующей странице).

Переключаться между кнопками можно, нажимая либо клавишу Tab, либо клавиши-стрелки. При переключении между кнопками перемещается пунктирная рамка внутри кнопки, указывая на то, что кнопка имеет фокус ввода. Когда кнопка имеет фокус ввода, весь клавиатурный ввод (кроме клавиш переключения между кнопками) направляется этой кнопке. Кнопка, имеющая фокус ввода, вызывает событие *Click* при нажатии пробела.

Кнопка по умолчанию обведена жирной линией — она получает нажатие клавиши Enter. Возможно, различие между кнопкой по умолчанию и кнопкой, имеющей фокус ввода, может показаться неочевидным. В этой программе это всегда одна и та же кнопка. Но по мере того, как вы будете знакомиться с другими >ле-

ментами управления, отличие станет яснее. Кнопка, имеющая фокус ввода, всегда является кнопкой по умолчанию, но при этом, если другой элемент управления получит фокус ввода, кнопка по умолчанию, реагирующая на нажатие клавиши Enter, может продолжать существовать. В диалоговых окнах обычно кнопками по умолчанию являются кнопки с надписями OK, Open (Открыть) или Save (Сохранить). Эта кнопка срабатывает, когда фокус ввода имеет другой элемент управления и пользователь нажимает Enter. Кроме того, кнопка Cancel (Отмена) обычно является *кнопкой отмены,* срабатывающей при нажатии клавиши Esc. Я подробнее рассмотрю связанные с этим моменты в главе 16.



Можно заставить кнопку сработать, нажав клавишу, соответствующую подчеркнутой букве: *L* соответствует кнопке «Larger» (Больше), а *S* — «Smaller» (Меньше). Кнопка вызывает событие *Click,* однако фокус ввода не изменяется.

В начале метода *ButtonOnClick* параметр *object* приводится к типу *Button.* Теперь, сравнив этот объект с объектами *btnLarger* и *btnSmaller,* которые конструктор сохранил в полях, метод может принять решение о дальнейших действиях. В зависимости от того, какая кнопка была нажата, размер окна либо увеличивается, либо уменьшается на 10%. Кроме того, окно сдвигается на *5%,* так что остается на одном и том же месте экрана,

Изменение размеров окна приводит к вызову метода *OnResize,* в котором кнопки перемещаются в новый центр клиентской области. Перемещение кнопок можно было бы осуществлять и в методе *Click* после вычисления размеров клиентской области, но в этом случае они бы не перемещались при ручном изменении размеров окна пользователем.

Является ли перемещение элементов управления в зависимости от размеров клиентской области обычным делом? Нет. Но чтобы обеспечить такую функциональность, нужно выйти за рамки того, что может предоставить Windows Forms Designer. Это еще одна причина, по которой приходится писать код вручную, - для достижения гибкости.

Внешний вид и выравнивание

По умолчанию, кнопки (и другие элементы управления) наследуют свойства *Font, ForeColor* и *BackColor* от своих предков. Если в программе создаются дочерние элементы управления, а затем изменяется шрифт, цвет текста или цвет фона, то

внешний вид дочерних элементов управления изменяется соответственно. Так, в программе TwoButtons можно вставить оператор:

```
BackColor = Color.Blue;
```

где угодно в конструкторе или даже в методе *ButtonOnClick*. Оператор изменит цвет фона формы на голубой, и цвет фона кнопок тоже станет голубым.

Но можно изменить свойства *Font, ForeColor* или *BackColor* для одной из кнопок:

```
btnSmaller.BackColor = Color.Red;
```

После выполнения этого оператора цвет фона кнопки *btnSmaller* изменится на красный. (Можно изменить либо свойство *ForeColor*, либо *BackColor* на *Color.Transparent*, либо любой другой частично или полностью прозрачный цвет.) Что же произойдет при выполнении следующего оператора?

```
BackColor = Color.Magenta;
```

Фон формы станет пурпурным, фон кнопки *btnLarger* — тоже, но фон кнопки *btnSmaller* останется красным.

Как это работает? Элемент управления отслеживает программно измененные свойства и не изменяет их при изменении соответствующего свойства предка. Следующие методы восстанавливают состояние элемента управления по умолчанию, т. е. наследуют эти свойства от предка:

Методы *Control* (выборочно)

```
void ResetFont()
void ResetForeColor()
void ResetBackColor()
```

Свойство *TextAlign*, заданное в классе *ButtonBase* определяет ориентацию текста на кнопке:

Свойства *ButtonBase* (выборочно)

Тип	Свойство	Доступ
ContentAlignment	*TextAlign*	Чтение/запись

Это свойство можно присвоить равным одному из значений перечисления *ContentAlignment*. Как ни странно, *ContentAlignment* определено в пространстве имен *System.Drawing*, хотя и не используется в сочетании ни с каким классом из этого пространства имен.

Перечисление *ContentAlignment*

Член	Значение
TopLeft	0x0001
TopCenter	0x0002
TopRight	0x0004
MiddleLeft	0x0010
MiddleCenter	0x0020

Перечисление *ContentAlignment* *(продолжение)*

Член	Значение
MiddleRight	0x0040
BottomLeft	0x0100
BottomCenter	0x0200
BottomRight	0x0400

Хотя эти значения представляют собой битовые флаги, данная особенность — наследие ранней версии Windows Forms. Не следует сочетать значения *ContentAlignment*! Для кнопок значением *TextAlign* по умолчанию является *MiddleCenter*, что не кажется очевидным до тех пор, пока кнопка не станет заметно больше, чем размещенный на ней текст.

Класс *ButtonBase* содержит еще одно свойство, влияющее на внешний вид кнопки:

Свойства *ButtonBase* (выборочно)

Тип	Свойство	Доступ
FlatStyle	*Flatstyle*	Чтение/запись

FlatStyle — это перечисление, определенное в пространстве имен *System.Windows.Forms*:

Перечисление *Flatstyle*

Член	Значение	Описание
Flat	0	Плоская, а не объемная
Popup	1	Объемная при прохождении над ней курсора мыши
Standard	2	Обычная объемная
System	3	Стандартная для ОС

Значением по умолчанию является *FlatStyle.Standard*.

Следующая программа рисует 4 кнопки всеми четырьмя стилями. Обратите внимание, как здесь используется статический метод *Enum.GetValues* для получения массива всех значений *FlatStyle*. Каждое из них позволяет задать свойства кнопки *FlatStyle* и *Text*,

ButtonStyles.cs

```
//---------------------------------------
// ButtonStyles.cs © 2001 by Charles Petzold
// ---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ButtonStyles: Form
{
     public static void Main()
     {
          Application.Run(new ButtonStyles());
     }
```

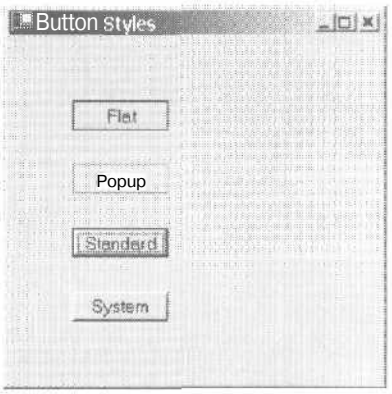
см. след. стр.

```
    public ButtonStyles()
    {
        Text = "Button Styles";

        int y = 0;

        foreach (FlatStyle fs in Enum.GetValues(typeof(FlatStyle)))
        {
            Button btn    = new Button();
            btn.Parent    = this;
            btn.FlatStyle = fs;
            btn.Text      = fs.ToString();
            btn.Location  = new Point(50, y += 50);
        }
    }
}
```

Вот результат работы этой программы:



Стиль Standard совпадает со стилем System, но в этом экранном снимке кнопка Standard (Стандартная) имеет фокус ввода и, таким образом, является кнопкой по умолчанию.

Кнопки с изображениями

В своем стремлении предоставить пользователю уникальный интерфейс не все удовлетворяются изменением шрифта и цветов на кнопках. Поместить на кнопку графическое изображение можно двумя способами. Первый — использовать четыре свойства класса *ButtonBase*:

Свойства *ButtonBase* (выборочно)

Тип	Свойство	Доступ
Image	*Image*	Чтение/запись
ImageList	*ImageList*	Чтение/запись
int	*ImageIndex*	Чтение/запись
ContentAlignment	*ImageAlign*	Чтение/запись

Эти свойства позволяют указать фоновое растровое изображение для кнопки. Можно либо указать для свойства *Image* конкретный объект *Image* или *Bitmap*, либо для свойства *ImageList*— объект *ImageList* и присвоить свойству *ImageIndex* значение индекса в списке изображений. Значение свойства *ImageAlign* по умолчанию равно *ContentAlignment.MiddleButton*.

Эти изображения вы можете получать любым способом: из ресурсов, файлов или формировать прямо в программе. Ширину и высоту кнопки следует указать равными ширине и высоте изображения плюс 8 (4 пиксела на каждую границу).

Чаще всего растровые изображения на кнопках применяются *вместо* текста, но одно не исключает другого. Если задать свойство *Text*, этот текст будет выводиться поверх изображения.

Далее приведена версия программы TwoButtons, которая загружает из ресурса два изображения размером 64x64 пиксела.

BitmapButtons.cs

```
//---------------------------------------------
// BitmapButtons.cs © 2001 by Charles Petzold
// -------- ------------------- --------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BitmapButtons: Form
{
     readonly int    cxBtn, cyBtn, dxBtn;
     readonly Button btnLarger, btnSmaller;

     public static void Main()
     {
          Application.fluri(newBitmapButtons());
     }
     public BitmapButtons( )
     {
          Text = "Bitmap Buttons";
          ResizeRedraw = true;

          dxBtn = Font.Height;

               // Создать первую кнопку.

          btnLarger = new Button();
          btnLarger.Parent = this;
          btnLarger.Image  = new Bitmap(GetType(),
                                   "BitmapButtons.LargerButton.bmp") ;

               // Вычислить размер кнопки на основе размера изображения.

          cxBtn = btnLarger.Image.Width  + 8;
```

см, след. стр.

```
            cyBtn - btnLarger.Image.Height + 8;

            btnLarger.Size  = new Size(cxBtn, cyBtn);
            btnLarger.Click += new EventHandler(ButtonLargerOnClick);

                // Создать вторую кнопку.

            btnSmaller = new Button();
            btnSmaller.Parent = this;
            btnSmaller.Image  = new Bitmap(GetType(),
                                    "BitmapButtons.SmallerButton.bmp");
            btnSmaller.Size   = new Size(cxBtn, cyBtn);
            btnSmaller.Click += new EventHandler(ButtonSmallerOnClick);

            OnResize(EventArgs.Empty);
        }
        protected override void OnResize(EventArgs ea)
        {
            base.OnResize(ea);

            btnLarger.Location =
                        new Point(ClientSize.Width / 2 - cxBtn - dxBtn / 2,
                                  (ClientSize.Height - cyBtn) / 2);
            btnSmaller.Location =
                        new Point(ClientSize.Width / 2 + dxBtn / 2,
                                  (ClientSize.Height - cyBtn) / 2);
        }
        void ButtonLargerOnClick(object obj, EventArgs ea)
        {
            Left   -= (int)(0.05 * Width);
            Top    -= (int)(0.05 * Height);
            Width  += (int)(0.10 * Width);
            Height += (int)(0.10 * Height);
        }
        void ButtonSmallerOnClick(object obj, EventArgs ea)
        {
            Left   += (int)(Width / 22f);
            Top    += (int)(Height / 22f);
            Width  -= (int)(Width / 11f);
            Height -= (int)(Height / 11f);
        }
    }
```

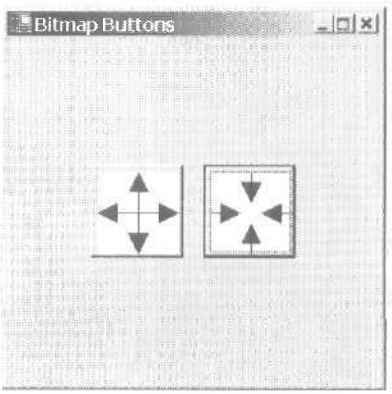
LargerButton.bmp



SmallerButton.bmp



В программе вычисляются величины *cxBtn* и *cyBtn* на основе размеров растрового изображения плюс 8. После создания каждой кнопки конструктор загружает растровое изображение из ресурса и устанавливает свойство кнопки *Image*. Вот как выглядят кнопки:



Несколько обработчиков или один?

В программе TwoButtons события *Click* от обеих кнопок обрабатывает единственный метод, в BitmapButtons — два отдельных обработчика. Разрабатывая формы и диалоговые окна с большим количеством элементов управления, вы, конечно, задумаетесь: иметь ли один обработчик для нескольких элементов или свой обработчик для каждого элемента?

Ни тот, ни другой подход абсолютно верным не назвать. Скорее всего код, в котором для каждого элемента управления используется отдельный обработчик, будет более ясным, и его будет легче сопровождать. С другой стороны, если обработчики для разных элементов управления исполняют одинаковый код, возможно, лучше объединить обработчики в один,

Прорисовка владельцем

Указание растрового изображения — не единственный способ отображать нестандартные кнопки. Можно полностью переопределить ее прорисовку, установив обработчик события *Paint* кнопки. Иногда такой подход называют *прорисовкой владельцем* (owner draw): ваша программа является владельцем кнопки, и именно она, а не сама кнопка ее прорисовывает.

Прорисовка владельцем не так проста, как применение растровых изображений, — ее лучше использовать, если изображением является простой векторный рисунок, просто потому, что векторная графика лучше масштабируется.

Прорисовку владельцем облегчает то, что совсем не нужно самостоятельно прорисовывать каждую деталь. Пространство имен *System.Windows.Forms* содержит

класс *ControlPaint,* включающий набор статических методов для прорисовки фрагментов общеупотребительных элементов управления. Так, для прорисовки кнопок могут пригодиться статические перегружаемые методы:

Статические методы *ControlPaint* (выборочно)

```
void DrawButton(Graphics grfx, int x, int y, int cx, int cy, ButtonState bs)
void DrawButton(Graphics grfx, Rectangle rect, ButtonState bs)
void DrawFocusRectangle(Graphics grfx, Rectangle rect)
void DrawFocusRectangle(Graphics grfx, Rectangle rect, Color clr, Color clr)
```

Метод *DrawButton* на самом деле просто рисует границу вокруг кнопки. Перечисление *ButtonState* представляет собой набор битовых флагов, управляющих состоянием КНОПКИ:

Перечисление *ButtonState*

Член	Значение
Normal	0x0000
Inactive	0x0100
Pushed	0x0200
Checked	0x0400
Flat	0x4000
Mi	0x4700

Я использую оба этих метода в программе OwnerDrawButtons.

OwnerDrawButtons.cs

```
//-------------------------------------------
// OwnerDrawButtons.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class OwnerDrawButtons: Form
{
     readonly int    cxImage, cyImage;
     readonly int    cxBtn, cyBtn, dxBtn;
     readonly Button btnLarger, btnSmaller;

     public static void Main()
     {
          Application.Run(new OwnerDrawButtons());
     }
     public OwnerDrawButtons()
     {
          Text = "Owner-Draw Buttons";
```

см. след. стр.

```
        ResizeRedraw = true;

        cxImage - 4 * Font.Height;
        cyImage = 4 * Font.Height;
        cxBtn = cxImage + 8;
        cyBtn * cylmage + 8;
        dxBtn = Font.Height;

        btnLarger - new Button();
        btnLarger.Parent = this;
        btnLarger.Size   = new Size(cxBtn, cyBtn);
        btnLarger.Click += new EventHandler(ButtonLargerOnClick);
        btnLarger. Paint += new PaintEventHandler(ButtonOnPaint);

        btnSmaller * new Button();
        btnSmaller.Parent = this;
        btnSmaller.Size   - new Size(cxBtn, cyBtn);
        btnSmaller.Click += new EventHandler(ButtonSmallerOnClick);
        btnSrnaller. Paint += new PaintEventHandler(ButtonOnPaint);

        OnResize(EventArgs.Empty);
    }
    protected override void OnResize(EventArgs ea)
    {
        base.OnResize(ea);

        btnLarger.Location =
                        new Point(ClientSize.Width / 2 - cxBtn - dxBtn / 2,
                                (ClientSize.Height - cyBtn) / 2);
        btnSmaller.Location =
                        new Point(ClientSize.Width / 2 + dxBtn / 2,
                                (ClientSize.Height - cyBtn) / 2);
    }
    void ButtonLargerOnClick(object obj, EventArgs ea)
    {
        Left   -= (int)(0.05 * Width);
        Top    -= (int)(0.05 * Height);
        Width  += (int)(0.10 * Width);
        Height += (int)(0.10 * Height);
    }
    void ButtonSmallerOnClick(object obj, EventArgs ea)
    {
        Left   += (int)(Width  / 22f);
        Top    += (int)(Height / 22f);
        Width  -= (int)(Width  / 11f);
        Height -= (int)(Height / 11f);
    }
    void ButtonOnPaint(object obj, PaintEventArgs pea)
    {
        Button   btn  = (Button) obj;
```

см. след. стр.

```
        Graphics grfx = pea.Graphics;

        ControlPaint.DrawButton(grfx, 0, 0, cxBtn, cyBtn,
             (btn == (Button) GetChildAtPoint(
                                  PointToClient(
                                       MousePosition))) &&
                  btn.Capture ? ButtonState.Pushed : ButtonState.Normal);

        GraphicsState grfxstate = grfx.Save();

        grfx.TranslateTransform((cxBtn - cxImage) / 2,
                                (cyBtn - cyImage) / 2);
        if (btn == btnLarger)
             DrawLargerButton(grfx, cxImage, cyImage);
        else
             DrawSmallerButton(grfx, cxImage, cyImage);

        grfx.Restore(grfxstate);

        if (btn.Focused)
             ControlPaint.DrawFocusRectangle(grfx,
                  new Rectangle((cxBtn - cxImage) / 2 + cxImage / 16,
                                (cyBtn - cyImage) / 2 + cyImage / 16,
                                7 * cxImage / 8, 7 * cyImage / 8));
   }
   void DrawLargerButton(Graphics grfx, int cx, int cy)
   {
        Brush brush = new SolidBrush(btnLarger.ForeColor);
        Pen   pen   = new Pen(btnLarger.ForeColor);

        grfx.TranslateTransform(cx / 2, cy / 2);

        for (int i = 0; i < 4; i++)
        {
             grfx.DrawLine(pen, 0, 0, cx / 4, 0);
             grfx.FillPolygon(brush, new Point[] {
                                  new Point(cx / 4, -cy / 8),
                                  new Point(cx / 2,      0),
                                  new Point(cx / 4,  cy / 8)});
             grfx.RotateTransform(90);
        }
   }
   void DrawSmallerButton(Graphics grfx, int cx, int cy)
   {
        Brush brush = new SolidBrush(btnSmaller.ForeColor);
        Pen   pen   = new Pen(btnSmaller.ForeColor);

        grfx.TranslateTransform(cx / 2, cy / 2);

        for (int i = 0; i < 4; i++)
```

см. след. с, пр.

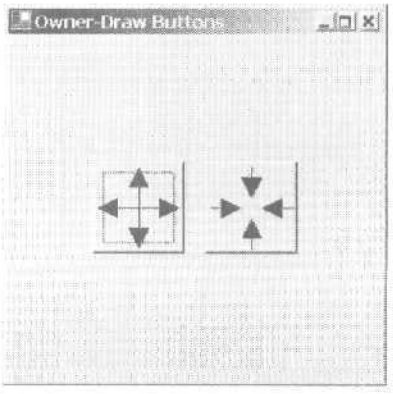
```
        {
            grfx.DrawLine(pen, 3 * cx / 8, 0, cx / 2, 0);
            grfx.FillPolygon(brush, new Point[] {
                                    new Point£3 * cx / 8, -cy / 8),
                                    new Pointf   cx / 8,       0),
                                    new Point(3 * ex / 8,   cy / 8)});
            grfx.RotateTransform(90);
        }
    }
}
```

Для обеих кнопок я устанавливаю обработчик события *Paint* с именем *Button-On Paint*. В начале обработчика вызывается *DrawButton*. Длинное выражение в середине вызова метода определяет, каким должен быть флаг, передаваемый в качестве последнего аргумента: *ButtonStateNormal* или *ButtonStatePushed*. Посмотрев на обычную кнопку, вы обнаружите, что она переключается в нажатое состояние, когда вы щелкаете ее. Кнопка остается в этом состоянии, пока кнопка мыши не будет отпущена или указатель мыши не покинет кнопку. Если снова поместить указатель на кнопку, все еще не отпуская кнопку мыши, то кнопка вернется в нажатое состояние. Реализованная в этой программе логика заключается в проверке свойства *Capture* кнопки (равно ли оно *true*), а также находится ли при этом указатель над кнопкой. Это работает!

Таким способом метод разбирается с прорисовкой границы. Затем вызываются методы *DrawLargerButton* и *DrawSmallerButton*, чтобы прорисовать внутренние области. Однако перед ними вызывается метод *TranslateTransform*, чтобы сдвинуть точку привязки графики внутрь границы. Я обрамляю перенос точки привязки вызовами методов *Save* и *Restore* объекта *Graphics*, чтобы они не влияли на вызов *DrawFocusRectangle*.

Методы *DrawLargerButton* и *DrawSmallerButton* прорисовывают внутренние области. Заметьте: сначала они устанавливают точку привязки в центр изображения, а затем рисуют одно и то же изображение 4 раза, каждый раз поворачивая его на 90%. (Вообще-то для рисования растровых изображений из BitmapButtons я использовал эти же два метода.)

Обработка *ButtonOnPaint* завершается вызовом *DrawFocusRectangle*. если кнопка имеет фокус ввода. Этот метод рисует обычную прямоугольную пунктирную рамку,



Бросаем якорь

Все приведенные ранее версии программы TwoButtons перемещали две кнопки в центр клиентской области каждый раз при изменении размеров клиентской области. Возможно, иногда вам бы хотелось, чтобы элементы управления динамически изменяли свое положение и размер в зависимости от размера клиентской области, но вам совсем не хочется самостоятельно писать обработчик события *OnResize*. Вам повезло: Windows Forms поддерживает два свойства элементов управления, динамически их перемещающих (и даже изменяющих их размер) — *Anchor* и *Dock*:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
AnchorStyles	*Anchor*	Чтение/запись
DockStyle	*Dock*	Чтение/запись

Эти свойства легко перепутать! В некоторых отношениях они похожи, и перечисления *AncborStyles* и *DockStyle* практически идентичны. Но эффект от их применения разный.

Перечисление *AnchorStyles*

Член	Значение
None	0
Top	1
Bottom	2
Left	4
Right	8

Эти значения представляют собой одиночные биты. Значения перечисления *AnchorStyles* можно объединять, используя оператор «поразрядное ИЛИ» языка C#.

Свойство *Anchor* устанавливается для элемента управления, а не для формы. При изменении размера формы оно определяет, от какой стороны или сторон формы элемент управления должен находиться на постоянном расстоянии.

По умолчанию это свойство не равно *AnchorStyle.None*! Значение по умолчанию — 5, что может выражаться как:

```
AnchorStyles.Left | AnchorStyles.Top
```

Смысл значения по умолчанию в том, что при изменении размера формы элемент управления будет находиться на неизменном расстоянии от левого и верхнего краев формы — собственно, именно этого мы от него и ожидаем.

Перепишем программу TwoButtons, задействовав преимущества свойства *Anchor*.

TwoButtonsAnchor.cs

```
//-----------------------------------------
// TwoButtonsAnchor.cs © 2001 by Charles Petzold
//-----------------------------------------
```

см. след. стр.

```
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoButtonsAnchor: Form
I
     public static void Main()
     {
          Application.Run(new TwoButtonsAnchor());
     }
     public TwoButtonsAnchor()
     {
          Text = "Two Buttons with Anchor";
          ResizeRedraw = true;

          int cxBtn = 5 * Font.Height;
          int cyBtn = 2 * Font.Height;
          int dxBtn =     Font.Height;

          Button btn = new Button();
          'btn.Parent   = this;
          btn.Text     = "&Larger";
          btn.Location = new Point(dxBtn, dxBtn);
          btn.Size     = new Size(cxBtn, cyBtn);
          btn.Click   += new EventHandler(ButtonLargerOnClick);

          btn = new Button();
          btn.Parent   = this;
          btn.Text     * "&Smaller";
          btn.Location * new Point(ClientSize.Width  - cxBtn - dxBtn,
                                   ClientSize.Height - cyBtn - dxBtn);
          btn.Size     = new Size(cxBtn, cyBtn);
          btn.Anchor   = AnchorStyles.Right | AnchorStyles.Bottom;
          btn.Click   += new EventHandler(ButtonSmallerOnClick);
     }
     void ButtonLargerOnClick(object obj, EventArgs ea)
     {
          Left   -= (int)(0.05 * Width);
          Top    -= (int)(0.05 * Height);
          Width  += (int)(0.10 * Width);
          Height += (int)(0.10 * Height);
     }
     void ButtonSmallerOnClick(object obj, EventArgs ea)
     {
          Left   += (int)(Width  / 22f);
          Top    += (int)(Height / 22f);
          Width  -= (int)(Width  / 11f);
          Height -= (int)(Height / 11f);
     }
}
```

Важнейшее изменение — отсутствие метода *OnResize*. Но если вам потребуется вызвать метод *OnResize* в программе, в которой используется свойство *Anchor*, не забудьте, что вызывать его надо так:

```
base.OnResize(ea);
```

иначе «заякоривание» элементов не сработает, Отказавшись от метода *OnResize*, я смог сделать переменные *cxBtn*, *cyBtn* и *dxBtn* локальными в конструкторе. Кроме того, так как я использую два разных обработчика события *Click*, нет нужды хранить два объекта «кнопка» в полях. Для создания обеих кнопок служит одна и та же переменная *btn*.

Кнопка Larger размещается в левом верхнем углу формы, а Smaller — в правом нижнем углу:



Кнопки не прижимаются к краям. Я использую переменную *dxBtn* (равную высоте шрифта) для задания расстояния между кнопками и границами клиентской области. Кнопка Larger сохраняет свойство *Anchor* по умолчанию, а для кнопки Smaller ему присваивается:

```
btn.Anchor = AnchorStyles.Right | AnchorStyles.Bottom;
```

Это означает, что Smaller будет оставаться на расстоянии *dxBtn* от правой и нижней границы клиентской области независимо от изменений ее размера. Если сделать клиентскую область очень маленькой, кнопки будут перекрываться,

Поэкспериментируем со стилями «якоря». Вот что мы обнаружим.

Если свойство *Anchor* содержит значения *AnchorStyles*, соответствующие противоположным сторонам, элемент управления изменяет свой размер при изменении размера формы. Так, если *Anchor* равно *AnchorStyle.Top* | *AnchorStyle.Bottom*, то ширина элемента управления не меняется, но его высота изменяется по мере изменения высоты формы. Так происходит оттого, что от края элемента управления до краев клиентской области сохраняется одинаковое расстояние. Если сделать форму слишком маленькой, элемент управления станет невидим.

Если установить *Anchor* так, чтобы оно соответствовало всем четырем сторонам, ширина и высота элемента управления будут изменяться по мере изменения размера формы.

Если *Anchor* равно одному из значений *AnchorStyle*, скажем, *AnchorStyle.Right*, расстояние между элементом управления и правой стороной клиентской области остается неизменным. Однако по мере изменения высоты формы, элемент

управления сохраняет свое положение по вертикали приблизительно пропорционально изменению формы.

Если установить *Anchor* равным *AnchorStyle.None*, элемент управления будет сохранять свое положение приблизительно пропорционально изменению формы. Так, если поместить элемент управления в центр клиентской области и установить его свойство ***Anchor*** в *AnchorStyle.None*, то по мере изменения размеров клиентской области размер элемента управления меняться не будет, при этом он будет оставаться примерно посредине клиентской области.

Производим стыковку

Теперь разберемся со стыковкой (docking). Вот значения перечисления *DockStyle*:

Перечисление *DockStyle*

Член	Значения
None	0
Top	1
Bottom	2
Left	3
Right	4
Fill	5

Во-первых, обратите внимание на то, что эти значения — не битовые флаги. Эти стили нельзя комбинировать вместе. Значением по умолчанию является *DockStyle.None*.

Если установить свойство *Dock* равным одному из четырех значений *DockStyle*, соответствующих сторонам, элемент управления будет прижат к этой стороне по всей длине, при этом элемент будет касаться двух прилегающих сторон. Например, если указать стиль *DockStyle.Top*, элемент управления будет находиться в самом верху клиентской области, а его ширина будет равна ширине клиентской области. При увеличении ширины формы ширина элемента управления тоже будет увеличиваться.

«Заякоривание» элемента обычно не вызывает изменения его размеров. Они изменяются, только если он привязан к противоположным сторонам и размер формы изменяется. Но если разместить элемент управления на одной из сторон, сделать его ширину или высоту равной длине этой стороны и указать свойство *Anchor* равным комбинации значений *AnchorStyles*, соответствующим этим трем сторонам, результат будет таким же, как если бы элемент управления был пристыкован к этой стороне.

Вот еще одна версия TwoButtons, где две кнопки пристыковываются к верхней и нижней сторонам:

TwoButtonsDock.cs

```
//-----------------------------------------
// TwoButtonsDock.cs © 2001 by Charles Petzold
//-----------------------------------------
```

см. след. стр.

```
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoButtonsDock: Form
{
     public static void Main()
     {
          Application.flun(new TwoButtonsDock());
     }
     public TwoButtonsDock()
     {
          Text = "Two Buttons with Dock";
          ResizeRedraw = true;

          Button btn = new Button();
          btn.Parent = this;
          btn.Text   = "&Larger";
          btn.Height = 2 * Font.Height;
          btn.Dock   = DockStyle.Top;
          btn.Click += new EventHandler(ButtonLargerOnClick);

          btn - new Button();
          btn.Parent = this;
          btn.Text   = "&Smaller";
          btn.Height = 2 * Font.Height;
          btn.Dock   = DockStyle.Bottom;
          btn.Click += new EventHandler(ButtonSmallerOnClick);
     }
     void ButtonLargerOnClick(object obj, EventArgs ea)
     {
          Left   -= (int)(0.05 * Width);
          Top    -= (int)(0.05 * Height);
          Width  += (int)(0.10 * Width);
          Height += (int)(0.10 * Height);
     }
     void ButtonSmallerOnClick(object obj, EventArgs ea)
     {
          Left   += (int)(Width  / 22f);
          Top    += (int)(Height / 22f);
          Width  -= (int)(Width  / 11f);
          Height -= (int)(Height / 11f);   .
     }
}
```

Здесь я отказался от переменных *cxBtn*, *cyBtn* и *dxBtn* и просто устанавливаю свойство *Height* каждой кнопки вдвое больше высоты шрифта по умолчанию. Свойство *Dock* располагает кнопку у указанной стороны и делает ее ширину равной длине стороны;



Пристыковывание редко используют для кнопок! Гораздо интересней пристыковывать панели инструментов (к верхней стороне) и строки состояния (к нижней). Я покажу эти методы в главе 20. Кроме того, пристыковка имеет смысл в программах вроде Windows Explorer, в которых элемент управления «дерево» пристыкован к левой стороне клиентской области, элемент управления «список» – справа, а посредине — разделитель, Такие программы мы рассмотрим в главе 22.

А что, если пристыковать два элемента управления к одной стороне? Вам, верно, будет приятно узнать, что при пристыковке к одной стороне элементы *не* перекрываются — они выстраиваются в «очередь» к краю. Так, если в программе TwoButtonsDock присвоить свойству *Dock* двух кнопок значение *DockStyle.Top*, кнопки буду выглядеть так:



Кажется, что позже созданный элемент управления расположен ближе к краю. На самом деле расположение элементов управления основано на понятии *z-порядка* (z-order), о котором я еще расскажу.

Теперь о *DockStyle.Fill*.Я использовал *DockStyle.Fill* в программе AnalogClock из главы 10, чтобы элемент управления «часы» занимал всю клиентскую область формы. Не следует задавать свойство *Dock*, равное *DockStyle.Fill* более чем для одного элемента управления. Элемент управления занимает всю клиентскую область, но не перекрывает другие элементы управления, у которых *Dock* отлично от значения по умолчанию.

Помните: в свойствах *Anchor* и *Dock* нет ничего сверхъестественного! Всю их функциональность можно реализовать в методе *OnResize*.

Дочерние элементы формы

Класс *Control* содержит весьма полезное и удобное неизменяемое свойство *Controls:*

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
Control.ControlCollection	*Controls*	Чтение

Хотя свойство *Controls* определено в классе *Control,* его полезность проявляется только в классах-наследниках *Control,* являющихся родительскими для дру их элементов управления, например *Form.* Тип *Control.ControlCollection,* определенный для этого свойства, может выглядеть немного экзотически, но на самом деле это просто открытый класс *ControlCollection,* определенный в классе *Control.* В программе не нужно ссылаться на имя этого класса — только на свойство *Controls,*

Нечто подобное вы уже видели в конце главы 11. Класс *ImageList* содержит свойство *Images* типа *ImageList.ImageCollection.* А в главе 14 вы увидите, что класс *Menu* содержит свойство *MenuItems* типа *Menu.MenuItemCollection.*

Класс *Control.ControlCollection* реализует интерфейсы *IList, ICollection* и *IEnumerable* (определенные в пространстве имен *System.Collections*). В результате свойство *Controls* представляет собой весьма гибкий массив (в точности как класс *ArrayList,* рассмотренный в конце главы 8), в который можно добавлять элементы и удалять их. По сути свойство *Controls* является массивом всех дочерних элементов управления формы.

Допустим, вы создали три кнопки, *btn1, btn2* и *btn3.* Вы делаете эти кнопки дочерними элементами формы так:

```
btn1.Parent = this;
btn2.Parent = this;
btn3.Parent = this;
```

После исполнения этих операторов можно получить количество дочерних элементов управления, используя свойство *Count* свойства *Controls* формы. Если эти кнопки — единственные дочерние элементы формы, выражение:

```
Controls.Count
```

вернет 3. Для доступа к свойству *Controls* можно использовать индекс, как и для массива, Возвращаемый объект имеет тип *Control.* Так, оператор:

```
Control ctrl = Controls[1];
```

запишет в перемененную *Ctrl* объект *btn2.* Если вы знаете, что элемент управления является кнопкой, можно привести возвращаемое значение к типу *Button:*

```
Button btn = (Button) Controls[1];
```

Индекс предназначен только для чтения. Вам не удастся сделать что-то вроде:

```
Controls[1] = new Button();  // Не выполнится!
```

Дочерние элементы управления попадают в класс *Control.ControlCollection* автоматически, как только они становятся дочерними элементами формы. Кроме того, поместить элемент управления в этот набор можно одним из двух следующих методов:

Методы *Control.ControlCollection* (выборочно)

```
void Add(Control Ctrl)
void AddRange(Control[] actrl)
```

На первый взгляд выражение:

```
Controls.Add(btn1);
```

выглядит, как вызов статического метода *Add* из класса *Controls*. Это не так! *Controls* — это свойство *Form*, унаследованное от *Control*. Тип этого свойства — *Control.ControlCollection*, т. е. класс, в котором определен метод *Add*. Вызов метода *Add* эквивалентен:

```
btn1.Parent = this;
```

Выражение:

```
Controls.AddRange(new Control[] { btn1, btn2, btn3 });
```

эквивалентно трем приведенным ранее выражениям, в которых присваивалось свойство *Parent* кнопок. Кстати, очевидно, что выражение *AddRange* было бы короче, если бы три кнопки находились в массиве.

Можно удалять элементы управления из набора:

Методы *Control.ControlCollection* (выборочно)

```
void Remove(Control ctrl)
void RemoveAt(int iIndex)
void Clear()
```

Удаление элемента из набора не уничтожает его. Это эквивалентно установлению свойства *Parent* этого элемента в *null*. Метод *Clear* удаляет все элементы управления из набора.

При удалении элемента оставшиеся элементы управления переиндексируется так, что «дырок» не останется. Пропущенных индексов никогда не бывает — индексы всегда находятся в диапазоне от 0 до *Count-1*.

Можно получить индекс конкретного элемента управления:

Методы *Control.ControlCollection* (выборочно)

```
bool Contains(Control Ctrl)
int GetChildIndex(Control Ctrl)
int GetChildIndex(Control Ctrl, bool bThrowException)
```

Перед вызовом *GetChildIndex* уместно вызвать метод *Contains*, дабы выяснить, содержит ли набор этот элемент управления. Если нет, первая версия *GetChildIndex* генерирует исключение. Вторая версия не генерирует исключение при *bThrowException*, установленном в *false*. Если элемент не содержится в наборе, метод возвращает –1.

Можно присвоить элементу управления новый индекс:

Методы *Control.ControlCollection* (выборочно)

```
void SetChildIndex(Control ctrl, int iNewIndex)
```

И еще раз: остальные элементы переиндексируются так, что индексы идут последовательно, от 0 до числа, на единицу меньшего количества элементов. Если хотите присвоить элементу управления максимальный индекс, установите *iNewIndex* в –1.

Зачем менять индексы в наборе элементов управления? А затем, что индексы служат не только для простого доступа к элементам управления — они также определяют z-порядок элементов управления.

Z-порядок

Я уже ссылался на z-порядок группы элементов управления, являющихся дочерними элементами формы. Термин *z-порядок* относится к трехмерному пространству координат, в котором по осям *x* и *y* откладываются горизонтальные и вертикальные координаты, а ось *z* перпендикулярна экрану.

Очевидно, что z-порядок влияет на внешний вид перекрывающихся элементов управления, имеющих общий родительский элемент управления. Наблюдать z-порядок можно в программе TwoButtonsAnchor, если сделать окошко таким маленьким, чтобы кнопки перекрывались. Кнопка Larger оказывается поверх кнопки Smaller. Кроме того, z-порядок определяет последовательность, в которой располагаются элементы управления, пристыкованные к одному краю формы,

Z-порядок зависит от последовательности присвоения свойству *Parent* элемента управления или добавления элементов в набор элементов управления. Z-порядок создается программно и может быть изменен только программно. Щелчки пользователя элементов управления его *не изменяют.*

Я сам часто путаю, что такое «вершина» и «низ» z-порядка, так что давайте четко это определим. Свойства элемента управления на *вершине* z-порядка таковы:

- у этого элемента управления свойство *Parent* было установлено *первым*, или это первый элемент, добавленный в набор элементов управления;
- его индекс в свойстве *Controls* равен 0;
- визуально он находится поверх всех других элементов управления; этот элемент получает события от мыши, когда ее указатель находится над ним, даже если то же место занимают другие элементы;
- этот элемент управления находится ближе всего к центру клиентской области, когда несколько элементов пристыкованы к одной и той же стороне контейнера.

Свойства элемента управления, находящегося *внизу* z-порядка, таковы:

- у этого элемента управления свойство *Parent* было установлено *последним* ЕЛИ это последний элемент, добавленный в набор элементов;
- его индекс в наборе элементов управления — *(ControlsCount - 1)*;
- он визуально находится ниже всех других элементов управления;
- этот элемент управления находится на краю контейнера, даже если к этому краю пристыкованы несколько элементов.

Помимо метода *SetChildIndex*, реализованного в классе *Control.ControlCollection*, контейнер может изменять z-порядок дочерних элементов, используя:

Методы *Control* (выборочно)

Метод	Описание
void `BringToFront()`	Помещает элемент управления на вершину z-порядка.
void `SendToBack()`	Помещает элемент управления вниз z-порядка.

Допустим, у формы есть три дочерних элемента управления: *btn1*, *btn2* и *btn3*. При этом *btn1* — на вершине, а *ЬIn3* — внизу z-порядка. Тогда метод:

```
btn3.BringToFront()
```

поместит *ЬIn3* на вершину z-порядка, а *btn2* вниз z-порядка. Это не то же, что:

```
btn1.SendToBack()
```

В этом случае *btn2* окажется на вершине z-порядка, а *btn1* внизу,

Флажок

Следующий элемент управления — флажок — состоит из маленького прямоугольника и текстовой строки. При щелчке элемента управления (или нажатии пробела, когда элемент имеет фокус ввода) в прямоугольнике появляется отметка. Если еще раз щелкнуть элемент управления, отметка исчезает. В отличие от кнопок флажок сохраняет свое состояние (включен/выключен),

Два важнейших свойства *CheckBox*:

Свойства *CheckBox* (выборочно)

Тип	Свойство	Доступ	Описание
bool	*Checked*	Чтение/запись	По умолчанию равно *false*
bool	*AutoCheck*	Чтение/запись	По умолчанию равно *true*

Свойство *Checked* определяет, отмечен ли элемент управления. Его можно использовать как для инициализации элемента, так и для выяснения его состояния. Скорее всего вам не придется менять значение по умолчанию свойства *AutoCheck*, так как при таком значении этого свойства элемент управления *CheckBox* сам меняет свое состояние при щелчках на нем,

Элемент управления вызывает событие *CheckedChanged* всякий раз при изменении свойства *Checked*.

События *CheckBox* (выборочно)

Событие	Метод	Делегат	Аргументы
CheckedChanged	*OnCheckedChanged*	*EventHandler*	*EventArgs*

Событие *CheckedCbanged* вызывается в двух случаях: когда пользователь щелкает элемент управления *CheckBox* при свойстве *AutoCheck*, установленном в *true*, и при программном изменении свойства *Checked* — например, при инициализации элемента.

Если вы установите *AutoCheck* в *false*, вам придется установить обработчик события *Click* элемента управления. Обработка события *Click* скорее всего будет содержать такие выражения, переключающие отметку:

```
CheckBox chkbox = (CheckBox) obj;
chkbox.Checked "= true;
```

Не переключайте свойство *Checked* в обработчике события *CheckedChanged*! Если же это сделать, каждое переключение будет снова вызывать событие *CheckedChanged*— до бесконечности...

Следующая программа создает 4 элемента управления *CheckBox*, позволяющих задать 4 атрибута шрифта: полужирный, курсив, подчеркнутый и перечеркнутый.

CheckBoxDemo.cs

```
//--------------------------------------
// CheckBoxDemo.cs в 2001 by Charles Petzold
//--------------.....-------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CheckBoxDemo: Form
{
    public static void Main()
    {
        Application.Run(new CheckBoxDemo());
    }
    public CheckBoxDemo()
    {
        Text = "CheckBox Demo";

        CheckBox[] achkbox  = new CheckBox[4];
        int        cyText   = Font.Height;
        int        cxText   - cyText / 2;
        string[]   astrText = {"Bold", "Italic", "Underline", "Strikeout"};

        for (int i - 6; i < 4; i++)
        {
            achkbox[i] = new CheckBox();
            achkbox[i].Text = astrText[i];
            achkbox[i].Location= new Point(2 * cxText,
                                           (4 + 3 * i) * cyText / 2);
            achkbox[i].Size = new Size{12 * cxText, cyText);
            achkbox[i].CheckedChanged +=
                            new EventHandler(CheckBoxOnCheckedChanged);
        }
        Controls.AddRange(achkbox);
    }
```

см. след. стр.

```
    void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
    {
        Invalidate(false);
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics    grfx = pea.Graphics;
        FontStyle   fs   = 0;
        FontStyle[] afs  = { FontStyle.Bold,      FontStyle.Italic,
                             FontStyle.Underline, FontStyle.Strikeout };

        for (int i = 0; i < 4; i++)
            if (((CheckBox) Controls[i]).Checked)
                fs |= afs[i];

        Font font = new Font(Font, fs);
        grfx.DrawString(Text, font, new SolidBrush(ForeColor), 0, 0);
    }
}
```

Я определил массив элементов управления *CheckBox* так, что я могу использовать метод *AddRange* свойства *Controls*. Переменная *cyText* содержит высоту свойства *Font* формы (и, соответственно, элемента управления). Я присваиваю переменной *cxText* вдвое меньшее значение, грубо вычисляя примерную ширину символов в нижнем регистре. Эти переменные позволяют задать свойства *Location* и *Size* каждого элемента управления. Хотя высота каждого элемента управления устанавливается в *cyText,* расстояние между элементами управления составляет 150% от этой величины. Ширины, равной *cxText*12,* хватает для текста и прямоугольника с отметкой. Вот как это выглядит;



В обработчике события *CheckedChanged* форма делается недействительной, что приводит к вызову *OnPaint*. Метод *OnPaint* осуществляет доступ по индексу к элементам свойства *Controls* формы и выясняет значение свойства *Checked* для каждого из четырех элементов управления и на основе этих данных создает переменную *FontStyle*. Теперь очень просто создать новый объект *Font* и вывести текст.

Будучи наследником *ButtonBase*, *CheckBox* обладает свойствами класса *Button*. Для флажков свойство *TextAlign* определяет расположение текста внутри прямоугольника, определенного свойством *Size*. Значение по умолчанию — *ContentAlignment.MiddleLeft*, т. е. текст центрирован по вертикали и выровнен по левому краю по горизонтали (но, конечно же, не заступает на сам прямоугольник с отметкой). Кроме того, класс *CheckBox* содержит свойство *CheckAlign*:

Свойства *CheckBox* (выборочно)

Тип	Свойство	Доступ
ContentAlignment	*CheckAlign*	Чтение/запись

Это свойство определяет местоположение прямоугольника с отметкой внутри прямоугольника. Значение по умолчанию — опять же *ContentAlignment.MiddleLeft*.

Чтобы поместить прямоугольник с отметкой справа от текста, стоит присвоить свойствам *TextAlign* и *CheckAlign* значение *ContentAlignment.MiddleRight*, и текст будет выровнен по правому краю. Интересно выглядит элемент управления, у которого высота примерно вдвое больше высоты шрифта, свойство *CheckAlign* установлено в *ContentAlignment.TopCenter*, а *TextAlign* — в *ContentAlignment.BottomCenter*. В таком элементе прямоугольник с отметкой горизонтально центрирован над горизонтально центрированным текстом.

Еще одно свойство класса *CheckBox* влияет на внешний вид элемента управления:

Свойства *CheckBox* (выборочно)

Тип	Свойство	Доступ
Appearance	*Appearance*	Чтение/запись

Перечисление *Appearance* определено так:

Перечисление *Appearance*

Член	Значение
Normal	0
Button	1

Если установить это свойство равным *Button*, элемент управления *CheckBox* будет выглядеть как кнопка, сохраняющая свое состояние. Вам придется увеличить высоту элемента управления, чтобы включить границу кнопки.

Вариант с тремя состояниями

Обычно объект *CheckBox* находится в одном из состояний типа «включен/выключен», «да/нет», «1/0». Такой элемент управления понравился бы Джорджу Булю. Но бывают ситуации, когда одного бита мало, а двух много. Тогда можно установить флажок в третье состояние.

Когда это может понадобиться? Допустим, в текстовом процессоре (например, в диалоговом окне выбора шрифта) имеется элемент управления *CheckBox* с пометкой «Курсив». Если пользователь выделяет текст, не набранный курсивом, элемент управления *CheckBox* должен находиться в неотмеченном состоянии. Если

же текст набран курсивом, элемент управления должен быть отмечен. А если текст набран курсивом только частично? В таком случае у вас есть хороший кандидат на третье состояние. В прямоугольнике рисуется отметка, но при этом прямоугольник имеет светло-серый цвет.

Не путайте третье состояние с заблокированным элементом *CheckBox*. В нашем случае надо заблокировать *CheckBox,* если выделенный текст выводится шрифтом, для которого не предусмотрено курсивное начертание.

Для трехпозиционных флажков применяются два свойства:

Свойства *CheckBox* (выборочно)

Тип	Свойство	Доступ	Описание
bool	*ThreeState*	Чтение/запись	По умолчанию *false*
CheckState	*CheckState*	Чтение/запись	Используется вместо *Checked*

Во-первых, надо установить свойство *ThreeState* в *true*. После чего для инициализации элемента управления или определения его состояния вместо свойства *Checked* используйте свойство *CbeckState*. Перечисление *CheckState* содержит три значения:

Перечисление *CheckState*

Член	Значение
Unchecked	0
Checked	1
Indeterminate	2

Если вы хотите иметь возможность реагировать на изменения свойства *CbeckState,* не устанавливайте обработчик события *CheckedChanged*. Вместо этого установите обработчик события *CheckStateChanged*:

События *CheckBox*

Событие	Метод	Обработчик	Аргумент
CheckStateChanged	*OnCheckStateChanged*	*EventHandler*	*EventArgs*

По мере того как пользователь щелкает трехпозиционный *CheckBox,* элемент управления проходит через эти три состояния.

Элемент управления «надпись»

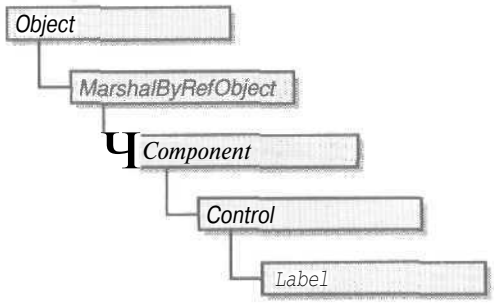
Иногда спрашивают, можно ли смешивать графику и элементы управления на одной и той же форме? Можно — программа CheckBoxDemo подтверждает это. Заметьте, что в ней метод *CheckBoxOnCheckedChanged* содержит единственное выражение:

```
Invalidate(false);
```

Параметр *false* означает, что области формы, занятые дочерними элементами управления, не будут делаться недействительными. Этот аргумент предотвращает ненужную перерисовку элементов управления.

Хотя можно смешивать элементы управления и графику на одной форме, обычно для вывода текста и другой графики используют дополнительные элементы

управления. В главе 4 я показал, как выводить текст на элементе *Panel*. Элемент *Label* специально создан для вывода текста. Родословная *Label* небогата:



Вот еще одна версия CheckBoxDemo, создающая для вывода того же текста элемент *Label*. Вместо того, чтобы делать недействительной форму и выводить текст в методе *OnPaint*, в этой версии в методе *CheckBoxOnCheckedChanged* создается новый шрифт и устанавливается в качестве свойства *Font* элемента управления *Label*.

CheckBoxWithLabel.cs

```
//-------------------------------------------------------------------
// CheckBoxWithLabel.cs © 2001 by Charles Petzold
//-------------------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CheckBoxWithLabel : Form
{
     Label label;

     public static void Main()
     {
          Application.Run(new CheckBoxWithLabel());
     }
     public CheckBoxWithLabel()
     {
          Text = "CheckBox Demo with Label";

          int      cyText   = Font.Height;
          int      cxText   = cyText / 2;
          string[] astrText = {"Bold", "Italic", "Underline", "Strikeout"};

          label = new Label();
          label.Parent   = this;
          label.Text     = Text + ": Sample Text";
          label.AutoSize = true;

          for (int i = 0; i < 4; i++)
          {
```

см. след. стр.

```
            CheckBox chkbox - new CheckBox();
            chkbox.Parent = this;
            chkbox.Text = astrText[i];
            chkbox.Location = new Point(2 * cxText,
                                        (4 + 3 * i) * cyText / 2);
            chkbox.Size = new Size(12 * cxText, cyText);
            chkbox.CheckedChanged +=
                        new EventHandler(CheckBoxOnCheckedChanged);
        }
    }
    void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
    {
        FontStyle   fs   = 0;
        FontStyle[] afs  - { FontStyle.Bold,      FontStyle.Italic,
                             fontStyle.Underline, FontStyle.Strikeout };

        for (int i = 0; i < 4; i++)
            if (((CheckBox) Controls[i t 1]).Checked)
                fs |= afs[i];

        label.Font = new Font(label.Font, fs);
    }
}
```

Если длина текста больше ширины элемента управления, элемент *Label* будет разбивать текст на несколько строчек. Однако при этом полосы прокрутки не появляются. *Label* так же может выводить только одну строку текста — в этом помогут свойства:

Свойства *Label* (выборочно)

Тип	Свойство	Доступ
int	*PreferredWidth*	Чтение
mt	*PreferredHeight*	Чтение
bool	*AutoSize*	Чтение/запись

Первые два свойства соответствуют данным, возвращаемым *MeasureString*, округленным до ближайшего наибольшего целого. Третье (по умолчанию равное *false)* позволяет сделать размер элемента «надпись» равным *PreferredWidth* и *PreferredHeight*.

Для вывода изображений элемент *Label* поддерживает те же четыре свойства, что и *ButtonBase: Image, ImageList, ImageIndex* и *ImageAlign*. Свойство *AutoSize не* изменяет размер элемента управления в зависимости от размера изображения.

Два дополнительных свойства влияют на внешний вид элемента *Label*:

Свойства *Label* (выборочно)

Тип	Свойство	Доступ
BorderStyle	*BorderStyle*	Чтение/запись
bool	*UseMnemonic*	Чтение/запись

Свойство *BorderStyle* определяет вид рамки вокруг надписи, его следует установить в одно из значений перечисления *BorderStyle*:

Перечисление *BorderStyle*

Член	Значение
None	0
FixedSingle	1
Fixed3D	2

Значение по умолчанию *BorderStyle.None*.

При свойстве *UseMnemonic*, установленном в *true* (по умолчанию), элемент управления *Label* удаляет амперсанты и подчеркивает следующую за амперсантом букву. В связи с этим возникает вопрос: раз элемент *Label* предназначен для вывода текста и изображений, зачем ему мнемоники? Как вы увидите далее, нужно это, чтобы перемещаться по элементам управления вроде полос прокрутки, регуляторов и текстовых окон без фиксированного текста.

Порядок перехода и остановки по клавише Tab

Перемещение между дочерними элементами управления осуществляется с помощью клавиши Tab либо клавиш-стрелок. Однако если попробовать использовать эти клавиши в программе CheckBoxWithLabel, вы увидите, что фокус ввода не попадает на элемент *Label*. Это вполне разумно: надписи не предназначены для обработки клавиатурного ввода, вот они и не получают клавиатурный фокус.

Свойство *TabStop* определяет возможность перемещения на конкретный элемент управления с помощью клавиши Tab:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
bool	*TabStop*	Чтение/запись
int	*TabIndex*	Чтение/запись

У кнопок свойство *TabStop* установлено в *true*; у надписей — *в false*.

Если *TabStop* установлено в *true*, свойство *TabIndex* определяет порядок, в котором Tab перемещает фокус с одного элемента на другой. *TabIndex* устанавливается при присвоении элементу управления родительского элемента, таким образом, изначально значения *TabIndex* совпадают с индексами z-порядка. Однако изменение z-порядка не влияет на *TabIndex*. С другой стороны, это свойство можно изменять независимо от z-порядка.

Если у двух элементов управления значения *TabIndex* совпадают, элемент с нижним значением z-порядка получает фокус раньше.

Идентификация элементов управления

В программе CheckBoxWithLabel определено два массива: один с текстовыми строками для четырех флажков, а второй содержит соответствующие значения перечисления *FontStyle*. Увы, эти массивы определены в разных областях программы. Если изменить порядок элементов в одном массиве, не изменив порядка в другом, программа будет работать неправильно.

Более того: сравните, как свойство *Controls* индексируется в методе *OnPaint* программы CheckBoxDemo и как оно индексируется в методе *CheckBoxOnChecked-Changed* программы CheckBoxWithLabel. В первой программе индексы находятся в диапазоне от 0 до 3- Во второй первым дочерним элементом формы является *Label,* и, таким образом, он имеет индекс 0. Элементы управления *CheckBox* индексируются с 1 до 4.

Если изменить в CheckBoxWithLabel конструктор, так чтобы элемент *Label* делался дочерним элементом формы после элемента *CheckBox,* программа будет работать неправильно. Я уж не говорю о том, что индексирование элементов массива *Controls* в зависимости от порядка их создания — плохой стиль программирования. Для программы с несколькими элементами управления это может быть не так ужасно. Но если элементов много, это может стать настоящим кошмаром.

Отслеживать все создаваемые элементы управления можно несколькими способами. Например, хранить объекты — элементы управления в полях, как в программе TwoButtons. Можно устанавливать отдельный обработчик событий для каждого элемента. Другой подход — использовать для уникальной идентификации свойства (или чего-то вроде этого) каждого элемента. Конечно, свойство *Text* обычно уникально идентифицирует элемент управления, и в обработчике события можно задействовать текст элемента в операторе *switch* для определения того, какой элемент вызвал событие. Но при этом, если вам когда-либо понадобится изменить текст элемента управления, вам придется менять как код, присваивающий свойство *Text* элемента, так и конструкцию *switch-case* в обработчике события.

Итак, для идентификации объекта свойства *Text* недостаточно. А что было бы достаточно? Класс *Control* содержит свойство, которое можно установить равным любому подходящему объекту:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
object	*Tag*	Чтение/запись

Это свойство специально предназначено для идентификации элементов управления. При создании элемента управления можно присвоить свойству *Tag* любой объект. К примеру, приведу частичное определение элемента управления *CheckBox,* предназначенного для выбора цвета:

```
chkbox.Text = "Magenta";
chkbox.Tag = Color.Magenta;
```

В обработчике события прежде всего следует получить элемент управления *CheckBox,* вызвавший событие:

```
CheckBox chkbox = (CheckBox) obj;
```

а затем привести свойство *Tag* к объекту *Color:*

```
Color clr = (Color) Tag;
```

Если *Tag* не является объектом *Color,* будет сгенерировано исключение. Пример использования свойства *Tag* я приведу в программе AutoScaleDemo чуть позже,

Что приятно в объектно-ориентированном программировании вообще (и в Windows Forms в частности), это то, что для идентификации элемента управления вы легко можете добавлять к нему все, что угодно.

В следующей программе создается новый класс — наследник *CheckBox*, предназначенный для вывода стилей шрифта. В новом классе добавляется всего одно поле — *fontstyle* типа *FontStyle*. Вы можете видеть (в конце листинга), что определение этого нового класса требует минимального кода.

CustomCheckBox.cs

```
//-----------------------------------------
// CustomCheckBox.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CustomCheckBox: Form
{
   .. public static void Main()
      {
          Application.Run(new CustomCheckBox());
      }
      public CustomCheckBox()
      {
          Text = "Custom CheckBox Demo";

          int      cyText = Font.Height;
          int     cxText = cyText / 2;
          FontStyle[] afs = { FontStyle.Bold,     FontStyle.Italic,
                              FontStyle.Underline, FontStyle.Strikeout };

          Label label    = new Label();
          label.Parent   * this;
          label.Text     * Text + ": Sample Text";
          label.AutoSize * true;

          for (int i = 0; i < 4; i++)
          {
               FontStyleCheckBox chkbox * new FontStyleCheckBox();
               chkbox.Parent = this;
               chkbox.Text = afs[i].ToString();
               chkbox.fontstyle - afs[i];
               chkbox.Location = rvew Point(2 « cxText,
                                            (4 + 3 * i) * eyText / 2);
               chkbox.Size - new Size(12 * cxText, cyText);
               chkbox.CheckedChanged +=
                              new EventHandler(CheckBoxOnCheckedChanged);
          }
      }
```

см. след. стр.

```
    void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
    {
        FontStyle fs = 0;
        Label     label = null;

        for (int i = 0; i < Controls.Count; i++)
        {
            Control ctrl = Controls[i];

            if (ctrl.GetType() == typeof(Label))
                label = (Label) ctrl;

            else if (ctrl.GetType() == typeof(FontStyleCheckBox))
                if (((FontStyleCheckBox) ctrl).Checked)
                    fs |= ((FontStyleCheckBox) ctrl).fontstyle;
        }
        label.Font = new Font(label.Font, fs);
    }
}
class FontStyleCheckBox: CheckBox
{
    public FontStyle fontstyle;
}
```

В конструкторе определяется массив значений *FontStyle*. При создании каждого объекта *FontStyleCheckBox* его полю *fontstyle* присваивается значение типа *FontStyle*. В программе не используется массив *string*. Вместо этого значение *FontStyle* преобразуется в строку, а свойству *Text* присваивается эта строка. Хотя массив строк по-прежнему необходим, по крайней мере оба массива определяются в одном фрагменте программы и при необходимости могут быть изменены одновременно.

Метод *CheckBoxOnCheckedChanged* циклически обрабатывает все элементы управления в массиве *Controls* и определяет тип каждого из них. Для определения типа объекта можно применять метод *GetType*, кроме того, можно использовать оператор *typeof* языка C# с любым именем класса для определения его типа. Если элемент управления имеет тип *FontStyleCheckBox*, объект приводится к этому типу, и если свойство *Checked* равно *true*, метод обращается к его полю *fontstyle*. Если элемент имеет тип *Label*, объект *Label* сохраняется, и обработка события завершается присвоением его свойству *Font* нового шрифта. Можно изменять значения *FontStyle* и порядок создания элементов и установления их родительской формы, при этом переделок кода не потребуется.

Автоматическое масштабирование

Программы из этой главы использовали свойство *FontHeight* формы, чтобы установить размер элементов управления с текстом. Для вычисления ширины элемента управления я брал половину высоты шрифта и считал ее примерно равной ширине символов в нижнем регистре. (Этого даже много для шрифта Courier.) Так как элементы управления наследуют шрифт от своих предков, этот способ отлично подходит для масштабирования элементов управления. Если нужно изменить

шрифт формы (и элементов), это следует сделать в конструкторе формы до получения высоты шрифта.

.NET Framework предлагает иной метод масштабирования элементов управления, *автоматическое масштабирование* (auto-scale). Для этого служат свойства класса *Form:*

Свойства *Form* (выборочно)

Тип	Свойство	Доступ	Описание
bool	*AutoScale*	Чтение/запись	**По умолчанию** *true*
Size	*AutoScaleBaseSize*	Чтение/запись	Высота и ширина шрифта формы

Единственный статический метод *Form* тоже связан с автоматическим масштабированием:

Статический метод Form

```
SizeF GetAutoScaleSize(Font font)
```

Свойство *AutoScaleBaseSize* и метод *GetAutoScaleSize* полезны сами по себе, даже если вы не применяете автоматическое масштабирование. В Windows Forms это единственный источник сведений о средней ширине символов конкретного шрифта. По умолчанию *AutoScaleBaseSize* возвращает ширину и высоту свойства *Font* формы, а *GetAutoScaleSize* — ширину и высоту любого объекта *Font.* Возвращаемая ширина вычисляется как средняя на основе символов латинского алфавита в нижнем регистре. Высота же равна свойству *Height* объекта *Font.*

Допустим, вы установили в параметрах экрана Small Fonts. В таком случае *AutoScaleBaseSize* сообщает, что шрифт Windows Forms по умолчанию в 8 пт имеет ширину 5 и высоту 13 пикселов. Для Large Fonts, *AutoScaleBaseSize* сообщает о ширине 6 пикселов и высоте 15. Так что, если хотите точнее позиционировать элементы управления, используйте:

```
cxText = AutoScaleBaseSize.Width;
cyText = AutoScaleBaseSize.Height;
```

Только что я сказал, *что AutoScaleBaseSize* возвращает ширину и высоту, вычисленные на основе свойства *Font* формы. Это так. И если изменить свойство *Font* формы, *AutoScaleBaseSize* вернет ширину и высоту нового шрифта. Однако это справедливо лишь в том случае, если сама форма не устанавливает свойство *AutoScaleBaseSize.* Если же форма сама установила это свойство (я имею в виду, что вы написали код, устанавливающий это свойство), оно возвращает значение, заданное формой.

Вот в чем секрет автоматического масштабирования: если форма устанавливает свойство *AutoScaleBaseSize,* ширина и высота дочерних элементов управления масштабируется в зависимости от отношения высоты и ширины, определяемой свойством *Font* формы к высоте и ширине, определяемых свойством *AutoScaleBaseSize.* Если форма не устанавливает *AutoScaleBaseSize,* эти отношения равны 1, и масштабирования не происходит.

Вообще во время установки свойств *Location* и *Size* дочерних элементов управления форма может использовать любую систему координат и любые размеры.

Затем с помощью *AutoScaleBaseSize* указывается система координат. Масштабирование происходит на основании текущих свойств *Font* и *AutoScaleBaseSize* после выполнения конструктора формы.

Поначалу автоматическое масштабирование выглядит сложным, так что рассмотрим несколько примеров.

Автоматическое масштабирование и Windows Forms Designer

Я уже упоминал, что однажды вам придется взглянуть на код, генерируемый Visual C#.NET Windows Forms Designer, и разобраться, что же там происходит. И вот вам пример — автоматическое масштабирование.

Допустим, для дисплея установлены Small Fonts. Когда вы конструируете форму с помощью Windows Forms Designer, он генерирует код, содержащий размер в пикселах. Например, свойство *Size* элемента управления *Button* можно установить так:

```
this.button1.Size = new System.Drawing.Size(104, 26);
```

Да, Windows Forms Designer многословен. Но суть не в этом. Обычно жесткое кодирование положения и размеров элементов управления приводит к неприятностям. Можете быть уверены, что рано или поздно кто-нибудь запустит вашу программу в режиме Large Fonts или с еще большими шрифтами и текст на кнопках будет обрезан. Но Windows Forms Designer добавляет еще одно выражение в конструктор:

```
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
```

Этот размер представляет собой ширину и высоту шрифта в режиме Small Fonts. По сути дела Windows Forms Designer вставляет в код разрешение экрана, на котором основано местоположение и размеры элементов управления.

Если эта программа запускается на компьютере, на котором задан режим Small Fonts, отношения высоты и ширины шрифта к высоте и ширине свойства *AutoScaleBaseSize* равно 1, а местоположение и размеры элементов управления применяются напрямую.

Если же на компьютере действует режим Large Fonts, ширина и высота шрифта равны 6 и 15 соответственно. Когда конструктор формы завершает работу, Windows Forms масштабирует горизонтальные координаты и размеры, используя множитель 6/5 (это средняя ширина, вычисленная из свойства *Font* формы, деленная на *AutoScaleBaseSize.Width*). Вертикальные координаты и размеры масштабируются с использованием множителя 15/13, что представляет собой высоту шрифта формы, деленную на *AutoScaleBaseSizeHeight*.

Именно так Windows Forms Designer избавляет от проблем, связанных с жестким заданием размеров и координат,

Вы и сами можете делать что-то в этом роде. Можно задавать все размеры и местоположение элементов управления так, чтобы они нормально выглядели на вашем компьютере, а затем добавить выражение, которое устанавливает свойство *AutoScaleBaseSize* в *Size*, соответствующий вашему компьютеру. Не забудьте проверить работу программы при разных размерах шрифта, изменяя либо размер шрифта в системе, либо задавая другое свойство *Font* в конструкторе,

Однако применение *AutoScaleBaseSize* может быть и более интересным.

Творческий подход к AutoScaleBaseSize

Обычно Windows-программисты, пишущие на C и C++, используя Win32 API или библиотеки Microsoft Foundation Class (MFC), определяют диалоговые окна в текстовом формате, называемом *шаблоном диалогового окна,* и применяют специальную независимую от устройства *систему координат диалогового окна.* Координаты *x* измеряются в 1/4 от средней ширины шрифта, координаты *y* — в 1/8 высоты шрифта. Так, если местоположение задано как (40, 32), элемент располагается на расстоянии 10 средних размеров символов по ширине от левого края диалогового окна и на расстоянии, равном высоте 4 символов от верхнего края.

В Windows Forms можно использовать ту же координатную систему. Все, что для этого нужно, — добавить следующее выражение в конструктор:

```
AutoScaleBaseSize = new Size(4, 8);
```

С другой стороны, можно указывать размеры и местоположение в целых единицах высоты и ширины символов. В таком случае вам остается добавить лишь:

```
AutoScaleBaseSize = new Size(1, 1);
```

Вообще-то такая система координат может показаться немного грубой, Обычно, чтобы экран выглядел более эстетично, расстояние между элементами управления по вертикали должно быть не меньше половины высоты шрифта.

Внутри автоматического масштабирования

После исполнения кода конструктора формы, она сама и ее дочерние элементы управления масштабируются на основании свойств *Font* к *AutoScaleBaseSize* формы. На самом деле масштабирование выполняется защищенным методом *ScaleCore* класса *Control,* вызываемом сначала для формы, а затем для всех дочерних элементов управления.

Можно самостоятельно выполнять масштабирование, используя методы *Scale:*

Методы *Control* (выборочно)

```
void Scale(float fScale)
void Scale(float xScale, yScale)
```

Например, установив свойство *AutoScale в false,* вы можете имитировать автоматическое масштабирование, вставив такие строки в конце конструктора формы:

```
Scale(GetAutoScaleSize(Font).Width  / AutoScaleBaseSize.Width,
      GetAutoScaleSize(Font).Height / AutoScaleBaseSize.Height);
```

Это ширина и высота *Font* формы, деленные на высоту и ширину, указанные в свойстве *AutoScaleBaseSize.*

Если нужно перемасштабировать элементы управления — к примеру, потому что вы изменили свойство *Font* где либо еще, кроме конструктора, — не полагайтесь на то, что автоматическое масштабирование выполнит эту работу за вас. Вам придется напрямую вызвать *Scale.* Имейте в виду: форма и элементы управления не хранят историю предыдущих масштабирований. После того, как элементы управления были масштабированы в конструкторе, их размеры и местоположения представляют собой значения в пикселах. Нельзя просто вызвать *Scale* с па-

раметрами, рассчитанными из свойств формы *Font* и *AutoScaleBaseSize*. Вам придется вычислить масштабирующий коэффициент на основе размера старого и нового шрифтов.

В следующей программе создается пять кнопок, позволяющих вам выбрать различные размеры шрифта. Конструктор программы использует размеры и местоположение на основе традиционной системы координат диалогового окна. Автоматическое масштабирование выполняет начальное масштабирование. При щелчке кнопки обработчик события *Click* снова масштабирует все на основе текущего и нового шрифтов.

AutoScaleDemo.cs

```
//---------------------------------------
// AutoScaleDemo.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class AutoScaleDemo: Form
{
     public static void Main()
     {
          Application.Run(new AutoScaleDemo());
     }
     public AutoScaleDemo()
     {
          Text = "Auto-Scale Demo";
          Font = new Font("Arial", 12);
          FormBorderStyle = FormBorderStyle.FixedSingle;

          int[] aiPointSize = { 8, 12, 16, 24, 32 };

          for (int 1 * 0; 1 < aiPointSize.Length; i++)
          {
               Button btn    = new Button();
              -fetia-,Parent  = this;
              ' btn.Text     = "Use " + aiPointSize[i] * "-point font";
               btn.Tag       = aiPointSize[i];
               btn.Location  = new Point(4, 18 + 24 * 1);
              ; btn.Size     = new Size(80, 18);
               btn.Click   += new EventHandler(ButtonOnClick);
          }
          ClientSize = new Slze(88, 16 + 24 * aiPointSize.Length);
          AutoScaleBaseSize = new Size(4, 8);
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          pea.Graphics.DrawString(Text, Font,
```

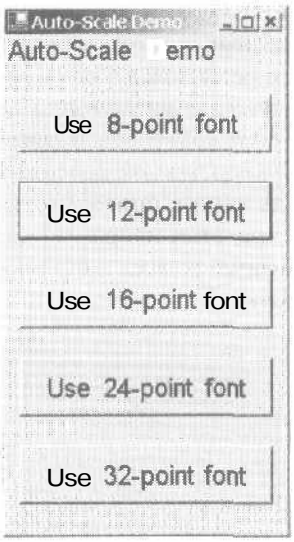
см. след. стр.

```
                                    new SolidBrush(ForeColor), 0, 0);
        }
        void ButtonOnClick(object obj, EventArgs ea)
        {
            Button btn = (Button) obj;

            SizeF sizefOld = GetAutoScaleSize(Font);
            Font = new Font(Font.FontFamily, (int) btn.Tag);
            SizeF sizefNew = GetAutoScaleSize(Font);

            Scale(sizefNew.Width  / sizefOld.Width,
                  sizefNew.Height / sizefOld.Height);
        }
}
```

Для хранения целочисленного значения размера в пунктах, связанного с каждой кнопкой, в программе используется свойство *Tag* элемента управления *Button*. В методе *ButtonOnClick* свойство *Tag* приводится к целому числу и создается шрифт. Вот окно программы при установленном размере шрифта формы 12 пт:



При щелчках кнопок размер формы изменяется согласно новому размеру шрифта,

Если создать элемент управления не в конструкторе формы, а где-то еще, придется вызвать метод *Scale* элемента управления с размерами текущего шрифта и свойством *AutoScaleBaseSize*.

Шестнадцатеричный калькулятор

Ниже приведена программа, создающая 29 элементов управления *Button* и реализующая шестнадцатеричный калькулятор с инфиксной записью. Программа HexCalc работает с 64-битными целыми числами без знака и может выполнять сложение, вычитание, умножение, деление, вычисление остатка, поразрядные И, ИЛИ, исключающее ИЛИ, а также поразрядный сдвиг. Вот как выглядит ее окно;



Работать с HexCalc можно с помощью мыши либо клавиатуры. Первым идет операнд (он вводится с помощью клавиатуры или кнопок), затем операция и второй операнд. Чтобы посмотреть результаты, надо щелкнуть кнопку Equals либо нажать клавиши «равно» (=) или Enter. Изменять введенные данные можно, щелкнув кнопку Back либо нажав клавишу Backspace. Окно ввода можно очистить, щелкнув его или нажав клавишу Esc.

В этой программе я не прибегал к Visual C# .NET Windows Forms Designer. Огромное количество кнопок с одинаковыми координатами и размерами прямо-таки требует более методического подхода. После пары безуспешных попыток я решил создать класс *CalcButton,* являющийся подклассом *Button,* и реализовал в нем конструктор, в качестве аргументов принимающий родительский элемент кнопки, ее текст, местоположение и размер. Я также добавил в него поле *chKey,* содержащее клавиатурный символ, связанный с этой кнопкой. Конструктор *HexCalc* содержит *29* операторов *new CalcButton,* создающих все кнопки. Для кнопок применяются традиционная система координат диалогового окна, но при этом напрямую вызывается короткая версия *Scale,* одинаково изменяющая масштаб во всех направлениях. Это позволяет сохранить большинство кнопок квадратными.

HexCalc.cs

```
//-------------------------------------
// NexGalc.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HexCalc : Form
{
     Button btnResult;
     ulong  ulNum       = 0;
     ulong  ulFirstNum  = 0;
     bool   bNewNumber  = true;
     char   chOperation = '=';

     public static void Main()
```

см. след. стр.

```
{
    Application.Run(new HexCalcO);
}
public HexCalcO
{
    Text * "Hex Calc";
    Icon = new Icon(GetType(), "HexCalc.HexCalc.ico");
    FormBorderStyle = FormBorderStyle.FixedSingle;
    MaximizeBox = false;

    new CalcButton(this, "D",      'D',  8,  24, 14, 14);
    new CalcButton(this, "A".      'A',  8,  40, 14, 14);
    new CalcButton(this, "7",      '7',  8,  56, 14, 14);
    new CalcButton(this, "4",      '4',  8,  72, 14, 14);
    new CalcButton(this, "1",      '1',  8,  88, 14, 14);
    new CalcButton(this, "0",      '0',  8, 104, 14, 14);
    new CalcButton(this, "E",      '£', 26,  24, 14, 14);
    new CalcButton(this, "8",      'B', 26,  40, 14, 14);
    new CalcButton(this, "8",      '8', 26,  56, 14, 14);
    new CalcButton(this, "5",      '5', 26,  T2, 14, 14);
    new CalcButton(this, "2",      '2', 26,  88, 14, 14);
    new CalcButton(this, "Back", '\x08', 26, 104, 32, 14);
    new CalcButton(this, "C",      'C', 44,  40, 14, 14);
    new CalcButton(this, "F",      'F', 44,  24, 14, 14);
    new CalcButton(this, "9",      '9', 44,  56, 14, 14);
    new CalcButton(this, "6",      '6', 44,  72, 14, 14);
    new CalcButton(this, "3",      '3', 44,  88, 14, 14);
    new CalcButton(this, "+", -    '+', 62,  24, 14. 14);
    new CalcButton(this, "-",      '-', 62,  40, 14, 14);
    new CalcButton(this, "*",      '*', 62,  56, t4, 14);
    new CalcButton(this, "/",      T,   62,  72, 14, 14);
    new CalcButton(this, "%",      T,   62,  88, 14, 14);
    new CalcButton(this, "Equals", '=', 62, 104, 32, 14);
    new CalcButton(this, "&&",     '&', 80,  24, 14, 14);
    new CalcButton(this, "|",      T.   80,  40, 14, 14);
    new CalcButton(this, "^",      '^', 80,  56, 14, 14);
    new CalcButton(this, "<",      '<', 80,  72, 14, 14);
    new CalcButton(this, ">",      '>', 80,  88, t4, 14);

    btnResult =
        new CalcButton(this, "0", '\x1B', 8, 4, 86, 14);

    foreach (Button frtn in Controls)
        btn.Click += new EventHandler(ButtonOnClick);

    ClientSize * new Size(102, 126);
    Scale(Font.Height / 8f);
}
protected override void OnKeyPress(KeyPressEventArgs kpea)
{
```

см. след. стр.

```
        char chKey = Char.ToUpper(kpea.KeyChar);

        if (chKey == '\x0D')
             chKey = '=';

        for (int i = 0; i < Controls.Count; i++)
        {
             CalcButton btn = (CalcButton) Controls[i];

             if (chKey == btn.chKey)
             {
                  InvokeOnClick(btn, EventArgs.Empty);
                  break;
             }
        }
   }
   void ButtonOnClick(object obj, EventArgs ea)
   {
        CalcButton btn = (CalcButton) obj;

        if (btn.chKey == '\x08')
             ulNum /= 16;

        else if (btn.chKey == '\x1B')
             ulNum = 0;

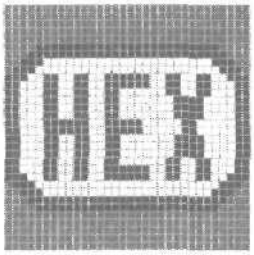
        else if (Char.IsLetterOrDigit(btn.chKey))     // Шестнадцатеричная
                                                      // цифра
        {
             if (bNewNumber)
             {
                  ulFirstNum = ulNum;
                  ulNum = 0;
                  bNewNumber = false;
             }
             if (ulNum <= ulong.MaxValue >> 4)
                  ulNum = 18 * ulNum +
                       (ulong)(btn.chKey -
                            (Char.IsDigit(btn.chKey) ? '0' : 'A' - 10»;
        }
        else                                          // Операция
        {
             if(!bNewNumber)
             {
                  switch(chOperation)
                  {
                  case '=*; ulNum = ulNum; break;
                  ease '+': ulNum = ulFirstNum + ulNum; break;
                  case '-': ulNum = ulFirstNum - ulNum; break;
                  case '*': ulNum = ulFirstNum * ulNum; fereak;
```

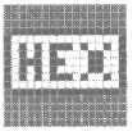
см. след. стр.

```
                case '&': ulNum = ulFirstNum & ulNum; break;
                case '|': ulNum = ulFirstNum | ulNum; break;
                case '^': ulNum = ulFirstNum ^ ulNum; break;
                case '<': ulNum = ulFirstNum << (int)ulNum; break;
                case '>*: ulNum = ulFirstNum >> (int)ulNum; frmak;
                case '/': ulNum - ulNum != б ?
                                ulFirstNum / ulNum : ulong.MaxValue;
                          break;
                case '%': ulNum = ulNum != 0 ?
                                ulFirstNum X ulNum : ulong.MaxValue;
                          break;
                default:  ulNum = 0; break;
                }
            }
            bNewNumber = true;
            chOperation = btn.chKey;
        }
        btnResult.Text = String.Format("{0:X}", ulNum);
    }
}
class CalcButton: Button
{
    public char chKey;

    public CalcButton(Control parent, string str, char chkey,
                      int x, int y, int ex, int cy)
    {
        Parent   = parent;
        Text     = str;
        chKey    = chkey;
        Location = new Point(x, y);
        Size     = new Size(cx, cy);
        SetStyle(ControlStyles.Selectable, false);
    }
}
```

HexCalc.ico





В HexCalc довольно хитро реализован клавиатурный интерфейс. Мне не хотелось, чтобы кнопки получали фокус ввода. Штриховая линия вокруг кнопки, обозначающая, что кнопка имеет фокус ввода, смотрелась не очень хорошо. Кроме

того, перемещать фокус ввода между кнопками посредством клавиши Tab, было бы бессмысленно. И наконец, клавиш, которые воспринимаются этой программой, больше, чем кнопок.

Чтобы переправить клавиатурные события форме, у каждой кнопки стиль *Selectable* установлен в *false,* благодаря чему кнопки не получают фокус ввода.

Метод *OnKeyPress* проходит по всем элементам массива *Controls* и находит кнопку, соответствующую нажатой клавише. Затем вызывается метод *InvokeOnClick*, имитируя событие кнопки *Click*. Таким образом, метод *ButtonOnClick* обрабатывает щелчки кнопок и нажатия их клавиатурных эквивалентов.

Переключатели и групповые блоки

Когда-нибудь все забудут, откуда взялось название кнопок-переключателей (radio buttons, радиокнопки). Видите ли, раньше на автомагнитолах помещали ряд кнопок, каждую из которых можно было настроить на какую-нибудь радиостанцию. Чтобы выбрать станцию, надо было нажать на кнопку, при этом нажатая ранее кнопка отщелкивалась. Так как одновременно может быть нажата только одна кнопка, набор кнопок-переключателей всегда представляет набор взаимоисключающих возможностей.

От других элементов управления переключатели отличает то, что они всегда объединены в группы. Так как одновременно может быть отмечена только одна кнопка-переключатель, состояние одного переключателя влияет на другие. Если отметить одну кнопку, отметка с другой снимется. Перемещение с помощью клавиш тоже в некотором роде нестандартно. Внутри группы кнопок-переключателей клавиши-стрелки переводят фокус ввода с одной кнопки на другую. При изменении фокуса меняется также и отмеченная кнопка-переключатель. Клавиша Tab позволяет перейти от группы переключателей к следующему элементу управления. При переходе с помощью клавиши Tab к группе переключателей отмеченный переключатель получает фокус ввода.

К счастью, большинство деталей реализации этого интерфейса скрыто от вас. Все, что вам нужно сделать, — это для каждой группы переключателей создать элемент управления *GroupBox* и сделать его дочерним элементом формы. Л потом сделать все объекты *RadioButton* в группе дочерними элементами *GroupBox*.

Сначала взглянем на класс *RadioButton*. Как и *CheckBox*, он содержит свойства *Checked* и *AutoCheck*:

Свойства *RadioButton* (выборочно)

Тип	Свойство	Доступ	Описание
bool	*Checked*	Чтение/запись	По умолчанию *false*
bool	*AutoCbeck*	Чтение/запись	По умолчанию *true*

Свойство *Checked* определяет, отмечен ли переключатель (это обозначает точка внутри крута). Свойство *AutoCheck*, установленное в *true,* автоматизирует использование кнопок-переключателей. Как только пользователь щелкает переключатель (или перемещает фокус ввода внутри группы), переключатели автоматически отмечаются/сбрасываются. Если присвоить *false* свойству *AutoCbeck* придется установить обработчик события *Click* и отмечать/сбрасывать переключатели самостоятельно.

Кроме этих свойств, в *RadioButton* определены еще только два открытых свойства: *Appearance* и *CheckAlign*, работающие так же, как и в классе *CheckBox*.

Событие *CheckedChanged* вызывается каждый раз, когда состояние кнопки меняется либо пользователем, либо программно:

События *RadioButton* (выборочно)

Событие	Метод	Делегат	Аргумент
CheckedChanged	*OnCheckedChanged*	*EventHandler*	*EventArgs*

События *CheckedChanged* вызываются парно: одно для кнопки, становящейся отмеченной, и еще одно — для сбрасываемой кнопки. Их можно различить, проанализировав свойство *Checked* переключателя. Во время события *CheckedChanged* свойство *Checked* соответствует новому состоянию.

Класс *GroupBox* является потомком *Control*, но реализует только одно открытое свойство *(FlatStyle)* и не определяет дополнительных по сравнению с *Control* методов и событий.

Рассмотрим пример. В следующей программе рисуется эллипс в зависимости от состояния восьми переключателей и одного флажка.

RadioButtons.cs

```
//-------------------------------------------
// RadioButtons.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RadioButtons: Form
{
     bool  bFillEllipse;
     Color colorEllipse;

     static void Main()
     {
          Application.Run(new RadioButtons());
     }
     RadioButtons()
     {
          Text = "Radio Buttons Demo";
          ResizeRedraw = true;

          string[] astrColor = { "Black", "Blue", "Green", "Cyan",
                                 "Red", "Magenta", "Yellow", "White"};

          GroupBox grpbox = new GroupBox();
          grpbox.Parent   = this;
          grpbox.Text     = "Color";
          grpbox.Location = new Point(Font.Height / 2, Font.Height / 2);
```

см. след. стр.

```
        grpbox.Size     = new Size(9 * Font.Height,
                                   (3 * astrColor.Length + 4) * Font.Height / 2);

        for (int i = 0; i < astrColor.Length; i++)
        {
            fladioButton radiobtn = new RadioButton();
            radiobtn.Parent      = grpbox;
            radiobtn.Text        = astrColor[i];
            radiobtn.Location    = new Point(Font.Height,
                                             3 * (i + 1) * Font.Height / 2);
            radiobtn.Size        = new Slze{7 * Font.Height,
                                            3 * Font.Height / 2);
            radiobtn.CheckedChanged +=
                                  new EventHandler(RadioButtonOnCheckedChanged);
            if(i ==O}
                radiobtn.Checked = true;
        }
        CheckBox chkbox = new CheckBox();
        chkbox.Parent   = this;
        chkbox.Text     = "fill Ellipse";
        chkbox.Location = new Point(Font.Height,
                                    3 * (astrColor.Length + 2) * Font.Height / 2);
        chkbox.Size     = new Size(Font.Height * 7, 3 * Font.Height / 2);
        chkbox.CheckedChanged +=
                              new EventHandler(CheckBoxOnCheckedChanged);
    }
    void RadioButtonOnCheckedChanged(object obj, EventArgs ea)
    {
        RadioButton radiobtn = (RadioButton) obj;

        if(radiobtn.Checked)
        {
            colorEllipse = Color.FromName(radiobtn.Text);
            Invalidate(false);
        }
    }
    void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
    {
        bFillEllipse = ((CheckBox)obj).Checked;
        Invalidate(false);
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx - pea.Graphics;
        Rectangle rect = new Rectangle(10 * Font.Height, 0,
                                       ClientSize.Width -
                                           10 * Font.Height - 1,
                                       ClientSize.Height - 1);
        if(bFillEllipse)
            grfx.FillEllipse(new SolidBrush(colorEllipse), rect);
```

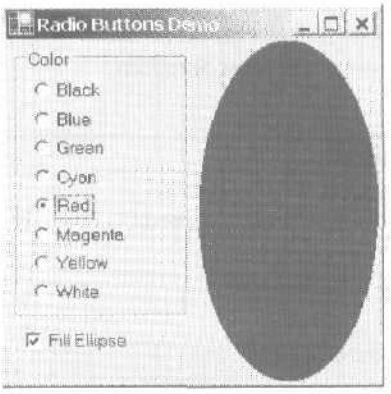
см. след. стр.

```
            else
                grfx.DrawEllipse(new Pen(colorEllipse), rect);
        }
    }
```

В начале конструктора определяется массив из 8 названий цветов. Все вертикальные координаты и размеры в программе вычисляются так, чтобы они вместили дополнительные названия цветов из этого массива (если это корректные названия цветов .NET Framework). (Однако ширина элементов управления недостаточна, чтобы вместить некоторые длинные названия.)

Сначала конструктор создает элемент управления *GroupBox*. Родительским элементом группового блока является форма. Затем конструктор создает 8 переключателей — дочерних элементов группового блока. Заметьте: в конце цикла *for* свойство *Checked* устанавливается в *true* для первой кнопки. Это выражение приводит к вызову *RadioButtonOnCheckedChanged*, в котором инициализируется поле *colorEllipse*. Конструктор завершается созданием элемента управления *CheckBox* — дочернего элемента формы.

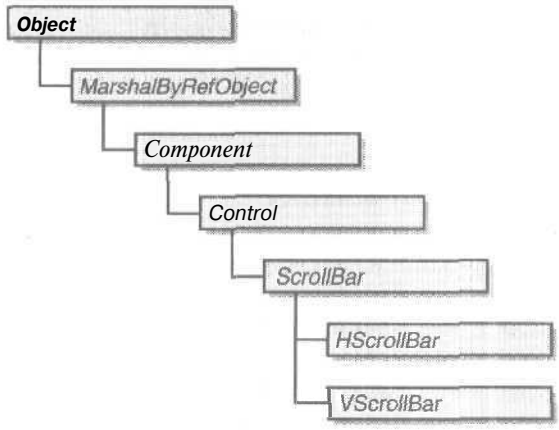
Клавиатурный интерфейс работает так, как я и описывал. При использовании клавиш-стрелок для перемещения между переключателями кнопки вызывают метод *RadioButtonOnCheckedChanged*, который использует статический метод *ColorFromName* и преобразует текст кнопки в объект *Color*. Как этот метод, так и *CheckBoxOnCheckedChanged* делают недействительной клиентскую область, что приводит к вызову *OnPaint*, рисующего эллипс:



Полосы прокрутки

В главе 4 мы обсуждали различия между элементом управления «полоса прокрутки» и полосами прокрутки, которые создаются автоматически и являются принадлежностью любого класса-наследника *ScrollableControl* (например, *Form* и *Panel*). При использовании автоматической прокрутки надо указать размер клиентской области, и полосы прокрутки автоматически появятся справа и внизу клиентской области. С автоматическими полосами прокрутки события не связаны — по крайней мере события, которые может получить приложение.

Класс *ScrollBar* является абстрактным классом-наследником *Control:*



У *ScroilBar* два наследника: горизонтальная полоса прокрутки (*HScrollBar*)и вертикальная полоса прокрутки *(VScrollBar)*. Их можно размещать в любом месте клиентской области и делать любого размера. Хотя вертикальные полосы прокрутки имеют ширину по умолчанию, а горизонтальные — высоту по умолчанию, вы можете создавать как очень тонкие, так и очень толстые полосы прокрутки. Однако для полос прокрутки нельзя установить фоновый цвет или цвет шрифта.

Чтобы использовать общие термины для вертикальных и горизонтальных полос прокрутки, я буду говорить о *толщине* и *длине*. Толщина — это высота горизонтальной полосы и ширина вертикальной, а длина — ширина горизонтальной и высота вертикальной. По умолчанию толщина полос прокрутки стандартна и равна *SystemInformation.VerticalScrollBarWidth* и *SystemInformation.HorizontalScrollBarHeight*.

Вот пять основных свойств, добавленных к *Control* классом *ScroilBar*.

Свойства *ScroilBar* (выборочно)

Тип	Свойство	Доступ	Описание
int	*Value*	Чтение/запись	Изменяется от *Minimum* до *(Maximum* + 1 - *LargeChange)*
int	*Minimum*	Чтение/запись	По умолчанию 0
int	*Maximum*	Чтение/запись	По умолчанию 100
int	*SmallChange*	Чтение/запись	По умолчанию 1
int	*LargeChange*	Чтение/запись	По умолчанию 10

Свойство *Value* представляет текущую позицию ползунка на полосе. Оно изменяется от *Minimum* до... ну, почти до *Maximum*. При щелчках стрелок на концах полосы свойство *Value* изменяется на *SmallChange*. При щелчке с любой стороны от ползунка свойство *Value* изменяется на *LargeChange*.

Почему же *Value* изменяется в диапазоне от *Minimum* до *(Maximum* + 1 - *LargeChange)*? Представьте документ, содержащий 500 строк текста. Свойство *Minimum* устанавливается 0 и *Maximum* в 499. Клиентская область может вместить 25 строчек текста. *SmallChange* присваивается 1 (одна строка текста), а *LargeChange — 25*.

Размер ползунка по отношению к длине полосы прокрутки зависит от отношения *LargeChange* к *Maximum*. Это доля видимой части документа.

Если *Value* равно 0, вы просматриваете начало документа, т. е. строки с 0 по 24, если номера строк в документе начинаются с 0. Когда *Value* равно 1, вы просматриваете строки с 1 по 25. И когда *Value* равно 475, вы просматриваете строки с 475 по 499. Это конец документа, и *Value* не должно более увеличиваться. Вот почему *Value* не становится больше, чем *(Maximum* + 1 - *LargeChange).*

Если вы не работаете с документами, вам придется установить *Maximum* так, чтобы *Value* находилось в необходимом диапазоне. Далее я приведу такой пример.

В *ScrollBar* реализованы два события;

События *ScrollBar*

Событие	Метод	Обработчик	Аргумент
ValueChanged	*OnValueChanged*	*EventHandler*	*EventArgs*
Scroll	*OnScroll*	*ScrollEventHandler*	*ScrollEventArgs*

Событие *ValueCbanged* происходит, когда на самом деле изменяется свойство *Value.* Если на вашу клавиатуру ляжет кошка, вам не придется тратить время на обработку огромного количества ненужных событий.

Событие *ValueChanged*происходит не только, когда пользователь что-то делает с полосой прокрутки, но и когда значение свойства *Value*изменяется программно. Событие *Scroll* не происходит, если свойство *Value* было изменено в программе.

Более того, событие *Scroll* может дать гораздо больше информации о том, что именно произошло с полосой прокрутки. Может, вам и не придется никогда использовать событие *Scroll,* но оно необходимо, если события *ValueCbanged* недостаточно. Обработчик события *Scroll* принимает объект типа *ScrollEventArgs,* имеющий такие свойства:

Свойства *ScrollEventArgs*

Тип	Свойство	Доступ
int	*NewValue*	Чтение/запись
ScrollEventType	*Type*	Чтение/запись

Свойство *NewValue* равно значению, которое будет присвоено свойству *Value* полосы прокрутки после того, как обработчик события вернет управление полосе прокрутки. Его можно переопределить, присвоив произвольное значение. Свойство *Type* имеет тип *ScrollEventType.*

Перечисление *ScrollEventType*

Член	Значение	Описание
SmallDecrement	0	Мышь: левая кнопка или верхняя стрелка Клавиатура: Клавиши «стрелка-влево» или «стрелка-вверх»
SmallIncrement	1	Мышь: правая кнопка или нижняя стрелка Клавиатура: Клавиши «стрелка-вправо» или «стрелка-вниз»
LargeDecrement	*2*	Мышь: щелчок в левой или верхней области Клавиатура: Клавиша Page Up
LargeIncrement	3	Мышь: щелчок в правой или нижней области Клавиатура: Клавиша Page Down
ThumbPosition	4	Мышь: отпускание кнопки на бегунке
ThumbTrack	*5*	Мышь: нажатие кнопки на бегунке или перемещение

см. след. стр.

Перечисление *ScrollEventType* *(продолжение)*

Член	Значение	Описание
First	6	Клавиатура: клавиша Home
Last	7	Клавиатура: клавиша End
EndScroll	8	Операция прокрутки завершена

К примеру, полоса прокрутки имеет фокус ввода, и вы нажимаете и отпускаете одну из клавиш-стрелок. Или щелкаете стрелку полосы прокрутки. Вначале вы получите событие *Scroll*, в котором поле *Type* равно *ScrollEventArgsType.SmallIncrement* или *ScrollEventArgsType.SmallDecrement*Затем вы получите событие *ValueChanged* и затем еще одно событие *Scroll,* в котором на этот раз поле *Type* будет равно *ScrollEventArgsType.EndScroll*.Если *sb* — объект типа *ScrollBar*, а *sea* — объект типа *ScrollEventArgs*, последовательность событий будет такова:

Событие	sb.Value	sea.Type	sea.NewValue
Scroll	*N*	*SmallIncrement*	*N* + 1
ValueChanged	*N* +1		
Scroll	*N* + 1	*EndScroll*	*N* + 1

Если держать нажатой кнопку мыши (или клавишу-стрелку), вы получите серию событий, заканчивающихся *EndScroll:*

Событие	sb. Value	sea. Type	sea.NewValue
Scroll	*N*	*SmallIncrement*	*N* + 1
ValueChanged	*N* + 1		
Scroll	*N* + 1	*SmallIncrement*	*N* + 2
ValueChanged	*N* + 2		
Scroll	*N* + 2	*SmallIncrement*	*N* + 3
ValueChanged	*N* + 3		
Scroll	*N* + 3	*SmallIncrement*	*N* + 4
ValueChanged	*N* + 4		
Scroll	*N* + 4	*EndScroll*	*N* + 4

Вы не будете получать событие *ValueChanged,* если *Value* достигнет своего максимального или минимального значения. При нажатии клавиши End в общем случае вы получите такую последовательность событий:

Событие	sb. Value	sea. Type	sea.NewValue
Scroll	*N*	*Last*	*sbMaximum*
ValueChanged	*sbMaximum*		
Scroll	*sbMaximum*	*EndScroll*	*sbMaximum*

Однако если бегунок при нажатии клавиши End уже находился в конце полосы прокрутки, последовательность событий будет такой:

Событие	sb.Value	sea.Type	sea.NewValue
Scroll	*sbMaximum*	*Last*	*sbMaximum*
Scroll	*sbMaximum*	*EndScroll*	*sbMaximum*

Если «схватить» бегунок мышью и перемещать его, последовательность событий будет такой:

Событие	*sb.Value*	sea. *Type*	*sea.NewValue*
Scroll	*N*	*ThumbTrack*	*N*
Scroll	*N*	*ThumbTrack*	*N* + 1
ValueCbanged	*N* + 1		
Scroll	*N* + 1	*TbumbTrack*	*N* + 2
ValueCbanged	*N* + 2		
Scroll	*N* + 2	*TbumbTrack*	*N* + 2
ValueCbanged	*N* + 3		
Scrol	*N* + 3	*TbumbTrack*	*N* + 3
ValueCbanged	*N* + 4		
Scrol	*N* + *A*	*TbumbTrack*	*N* + 4
Scrol	*N* + 4	*EndScroll*	*N* + 4

В зависимости от скорости, с которой вы перемещаете ползунок, вы можете получать события не со всеми значениями. И на самом деле необходимость установки обработчика события *Scroll* вместо события *ValueChanged* определяется тем, как программа должна реагировать на быстрое перемещение ползунка. Попробуйте «схватить» ползунок мышью и резко «подергать» его. Если ваша программа не может с этим справиться, возможно, стоит установить обработчик события *Scroll* вместо события *ValueChanged.* Например, можно игнорировать все значения *Type,* кроме *EndScroll.*

Программа ColorScroll использует три полосы прокрутки: Red (Красный), Green (Зеленый) и Blue (Голубой), — позволяющие выбирать сочетание цветов. Результирующий цвет устанавливается в качестве фонового цвета для формы. Чтобы все надписи и полосы прокрутки были видимы, элемент управления *Panel* белого цвета занимает половину клиентской области формы. Все прочие элементы управления — 3 полосы прокрутки и 6 надписей — являются дочерними элементами *Panel.*

ColorScroll.cs

```
//--------------------------------------------
// ColorScroll.cs © 2801 by Charles Petzold
//--------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorScroll: Form
{
     Panel         panel;
     Label[]       alabelName  = new Label[33;
     Label[]       alabelValue - new Label[3];
     VScrollBar[]  avscroll    = new VScrollBar[3];

     public static void Main()
```

см. след. стр.

```
{
     Application.Run(new  ColorScrollO);
}
public ColorScroll()
{
     Text = "Color Scroll";

     Color[] acolor = { Color.Red, Color.Green, Color.Blue };

          // Создать панель.

     panel = new PanelO;
     panel.Parent = this;
     panel.Location = new Point(0, 0);
     panel.BackColor = Color.White;

          // Пройти в цикле по трем цаетам.

     for (int i = 0; i < 3; i++)
     Л
          alabelName[i] = new Label();
          alabelName[i].Parent = panel;
          alabelName[i].ForeColor = acolor[i];
          alabelName[i].Text = "4" + acolor[i].ToKnownColor();
          alabelName[i].TextAlign = ContentAlignment.MiddleCenter;

          avscroll[i] * new VScrollBar();
          avscroll[i].Parent * panel;
          avscroll[i].SmallChange = 1;
          avscroll[i].LargeChange = 16;
          avscroll[i].Minimum  = 0;
          avscroll[i].Maximum = 255 + avscroll[i].LargeChange - 1;
          avscroll[i].ValueChanged +=
                         new EventHandler(ScrollOnValueChanged);
          avscroll[i].TabStop = true;

          alabelValue[i] = new LabelO;
          alabelValue[i].Parent = panel;
          alabelValue[i].TextAlign = ContentAlignment.MiddleCenter;
     }
     Color color = BackColor;
     avscroll[0].Value = color.R;  // Вызвать событие ValueChanged
     avscroll[1].Value = color.G;
     avscroll[2].Value = color.B;

     OnResize(EventArgs.Empty);
}
protected override void OnResize(EventArgs ea)
{
     base.OnResize(ea);
```

см. след, стр.

```
        int cx * ClientSize.Width;
        int cy = ClientSize.Height;
        int cyFont =  Font.Height;

        panel.Size = new Slze(ex / 2, cy);

        for (int i = 0; i < 3; 1++)
        I
            alabelName[i].Location = new Point(i * cx / 6, cyFont / 2);
            alabelName[i].Size = new Size(cx / 6, cyFont);

            avscroll[i].Location * new Point((4 * i + 1) * ex / 24,
                                             2 * cyFont);
            avscroll[i].Size = new Size(cx / 12, cy - 4 * cyFont);

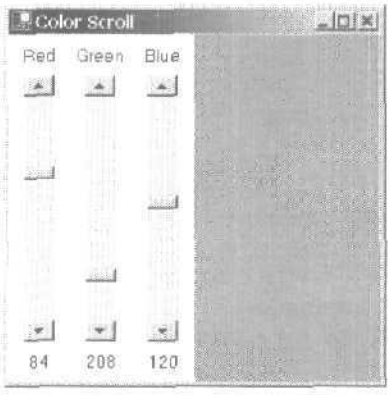
            alabelValue[i].Location = new Point(i * cx / 6,
                                                cy - 3 * cyFont / 2);
            alabelValue[i].Size = new Size(cx / 6, cyFont);
        }
    }
    void ScrollOnValueChanged(Object obj, EventArgs ea)
    {
        for (int i = 0; i < 3; i++)
            if((VScrollBar) obj == avscroll[i])
                alabelValue[i].Text = avscroll[i].Value.ToString();

        BackColor * Color.FromArgb(avscroll[0].Value,
                                   :   avscroll[1].Value,
                                       avscroll[2].Value);
    }
}
```

Конструктор создает все элементы управления и сохраняет их в полях. Полосы прокрутки должны предоставлять значения от 0 до 255. Заметьте: я устанавливаю *LargeChange* в 16 и затем присваиваю свойству *Maximum* значение 255 плюс *LargeChange* минус 1, что равно 270. При этом конструктор не определяет размеры и местоположение элементов управления. Это ответственное задание возложено на плечи метода *OnResize*. Местоположение и размеры элементов управления рассчитываются в зависимости от размера клиентской области и высоты шрифта. Вертикальные полосы прокрутки изменяют ширину при изменении размера формы. (Для достижения этого эффекта я пытался использовать «заякоривание», но у меня ничего не вышло.) Вот окно этой программы при обычных размерах (см. рис, на следующей странице).

Создаются два набора надписей: три элемента управления «надпись» хранится в массиве *alabelName*, а их свойствам *Text* заданы значения *Red, Green* и *Blue,* при этом их свойству *ForeColor* присвоен соответствующий цвет. Для этих целей я использую массив *acolor*. Если вызвать метод *ToString* объекта *Color,* результатом будет что-то вроде *Color [Red]*. Но если объект *Color* является членом перечисления

KnownColor, объект *Color* можно преобразовать в значение *KnownColor* методом *ToKnownColor*. Значение перечисления преобразуется в строки вида *Red*.



Обычно свойство *TabStop*, унаследованное от *Control*, для полос прокрутки устанавливается в *false*. В ColorScroll ему задано *true*. Кроме того, в начале текста надписей Red (Красный), Green (Зеленый) и Blue (Голубой) стоит амперсант. Так как фокус ввода на метки не переходит, нажатие подчеркнутой клавиши переводит фокус ввода на следующий элемент управления, способный получить фокус ввода. Таким образом, фокус на полосы прокрутки можно перевести как клавишей Tab, так и клавишами R, G или B.

При перемещении ползунков происходит событие *ValueChanged* и вызывается метод *ScrollOnValueChanged*. В этом методе параметр *obj* приводится к объекту *VscrollBar*, а затем осуществляется поиск в массиве *avscroll*. Индекс совпадающего элемента применяется для задания текста соответствующего элемента управления «надпись», отображающего значение (*alabelValue*). Метод завершается пересчитыванием цвета фона формы в зависимости от значений трех полос прокрутки.

Будьте осторожны при задании в программах свойства *Value* полосы прокрутки! Вначале конструктор ColorScroll завершался тремя выражениями, инициализирующими три полосы прокрутки значениями, составляющих цвета фона формы:

```
avscroll[0].Value = BackColor.R;
avscroll[1].Value = BackColor.G;
avscroll[2].Value = BackColor.B;
```

Однако первое же выражение вызывало событие *ValueChanged*, что приводило к вызову *ScrollOnValueChanged*, а в этом методе устанавливался цвет фона формы на основе значений *Value* трех полос прокрутки. Но так как полосы Green и Blue к этому моменту еще не инициализированы, то зеленая и голубая составляющая цвета фона — *BackColor.G* и *BackColor.B* — устанавливались в 0. Проблему решило сохранение цвета в отдельной переменной *Color* и использование ее для установки свойства *Value*:

```
Color color = BackColor;
avscroll[0].Value = color.R;
avscroll[1].Value = color.G;
avscroll[2].Value = color.B;
```

Регулятор как альтернатива полосам прокрутки

По своим функциям регулятор (track bar) очень похож на полосу прокрутки. С точки зрения программиста, единственная разница между ними заключается в том, что для регуляторов ориентация (горизонтальная или вертикальная) определяется свойством:

Свойства *TrackBar* (выборочно)

Тип	Свойство	Доступ
Orientation	*Orientation*	Чтение/запись

Определение перечисления *Orientation* коротко и ясно:

Перечисление *Orientation*

Член	Значение
Horizontal	0
Vertical	1

Эксперименты с программой CoJorScrotl показали, что толщину полос прокрутки можно менять. Толщину регуляторов по умолчанию изменять нельзя, да в этом обычно нет необходимости. Регулятору, как правило, нужна минимальная толщина для отображения отметок. Если хотите поэкспериментировать с толщиной регулятора, установите свойство *AutoSize* к *false*:

Свойства *TrackBar* (выборочно)

Тип	Свойство	Доступ	Описание
bool	*AutoSize*	Чтение/запись	По умолчанию *true*

По умолчанию свойство *AutoSize* равно *true,* а значит, ширина (для вертикальных регуляторов) или высота (для горизонтальных) будет неизменной независимс от свойства *Size*. По умолчанию для регуляторов свойство *TabStop*равно *true* (в отличие от полос прокрутки).

Класс *TrackBar* содержит те же свойства, что и *ScrollBar*, но значения по умолчанию для *Maximum* и *LargeChange* иные:

Свойства *TrackBar* (выборочно)

Тип	Свойство	Доступ	Описание
int	*Value*	Чтение/запись	Изменяется от *Minimum* до *Maximum*
int	*Minimum*	Чтение/запись	По умолчанию 0
int	*Maximum*	Чтение/запись	По умолчанию 10
int	*SmallChange*	Чтение/запись	По умолчанию 1
int	*LargeChange*	Чтение/запись	По умолчанию 5

Значение свойства *Value* для регуляторов просто и без затей изменяется от *Minimum* до *Maximum* без какой либо связи со свойством *LargeChange*. Это облегчает применение регуляторов в приложениях типа ColorScroll и затрудняет в приложениях, прокручивающих документы.

Хотя в вертикальных полосах прокрутки значения увеличиваются при сдвиге ползунка вниз, в вертикальных регуляторах значения увеличиваются при сдвиге ползунка вверх. Опять-таки существует разница между прокруткой документа и выбором значения.

Два дополнительных свойства позволяют управлять отметками на регуляторе:

Свойства *TrackBar* (выборочно)

Тип	Свойство	Доступ	Описание
TickStyle	*TickStyle*	Чтение/запись	По умолчанию *BottomRigbt*
int	*TickFrequency*	Чтение/запись	По умолчанию 1

С помощью свойства *TickStyle* можно указать, на какой стороне регулятора будут находиться отметки:

Перечисление *TickStyle*

Член	Значение	Описание
None	0	Без отметок
TopLeft	1	Отметки сверху для горизонтальных регуляторов и слева для вертикальных
BottomRigbt	2	Отметки снизу для горизонтальных регуляторов и справа для вертикальных
Both	3	Отметки с обеих сторон

Значением по умолчанию является *BottomRigbt*. Если установить *TickFrequency* в 1 (по умолчанию) и широкий диапазон для регулятора, все это может кончиться тем, что отметки сольются в сплошной черный прямоугольник.

Немного более гибкими регуляторы делает возможность указать цвет фона или фоновое изображение:

Свойства *TrackBar* (выборочно)

Тип	Свойство	Доступ
Color	*BackColor*	Чтение/запись
Image	*BackgroundImage*	Чтение/запись

Два важнейших события *TrackBar* называются так же, как и события, реализованные в *ScrollBar*:

События *TrackBar*

Событие	Метод	Делегат	Аргумент
ValueChanged	*OnValueChanged*	*EventHandler*	*EventArgs*
Scroll	*OnScroll*	*EventHandler*	*EventArgs*

Оба этих события делегированы обычному обработчику *EventHandler*. Для регуляторов события *Scroll* и *ValueChanged* всегда вызываются парами (сначала *Scroll,* затем *ValueChanged*), кроме тех случаев, когда свойству *Value* программно присваивается другое значение. В этом случае событие *ValueChanged* вызывается без соответствующего события *Scroll*

Вот программа ColorScroll, переписанная с использованием регуляторов.

ColorTrackBar.cs

```
//-----------------------------------------
// ColorTrackBar.cs © 2001 by Charles Petzold
//-----------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorTrackBar: Form
{
     Panel        panel;
     Label[]   alabelName  = new Label[3];
     Label[]   alabelValue = new Label[3];
     TrackBar[] atrackbar   = new TrackBar[3];

     public static void Main()
     i
          Application.Run(new ColorTrackBar());
     }
     public ColorTrackBar()
     {
          Text = "Color Track Bar";

          Color[] acolor = { Color.Red, Color.Green, Color.Blue };'

               // Создать панель.

          panel = new Panel();
          panel.Parent - this;
          panel.Location = new Point(0, 6);
          panel.BackColor = Color.White;

               // Пройти в цикле по трем цветам.

          for (int i = 0; i < 3; i++)
          {
               alabelName[i] - new Label();
               alabelName[i].Parent =  panel;
               alabelName[i].ForeColor = acolor[i];
               alabelName[i].Text - "&" + acolor[i].ToKnownColor();
               alabelName[i].TextAlign = ContentAlignment.MiddleCenter;

               atrackbar[i] = new TrackBar();
               atrackbar[i].Parent = panel;
               atrackbar[i].Orientation = Orientation.Vertical;
               atrackbar[i].BackColor - acolor[i];
               atrackbar[i].SmallChange = 1;
               atrackbar[i].LargeChange = 1S;
               atrackbar[i].Minimum = 0;
               atrackbar[i].Maximum - 255;
```

см. след. стр.

```
            atrackbar[i].TickFrequency = 16;
            atrackbar[i].ValueChanged +=
                        new EventHandler(TrackBarOnValueChanged);

            alabelValue[i]=new LabelO;
            alabelValue[i].Parent - panel;
            alabelValue[i].TextAlign = ContentAlignment.MiddleCenter;
        }
        Color color = BackColor;
        atrackbar[0].Value = color.R; // Вызвать событие ValueChanged
        atrackbar[1].Value = color.G;
        atrackbar[2].Value * color.B;

        OnResize(EventArgs.Empty);
    }
'   protected override void OnResize(EventArgs ea)
    {
        base.OnResize(ea);

        int cx = ClientSize.Width;
        int cy = ClientSize.Height;
        int cyFont = Font.Height;

        panel.Size = new Size(cx / 2, cy);

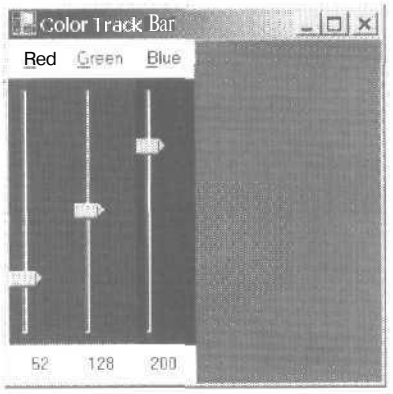
        for(inti=O;i<3; i++)
        {
            alabelName[i].Location = new Point(i * cx / 6, cyFont / 2);
            alabelName[i].Size = new Size(cx / 6, cyFont);

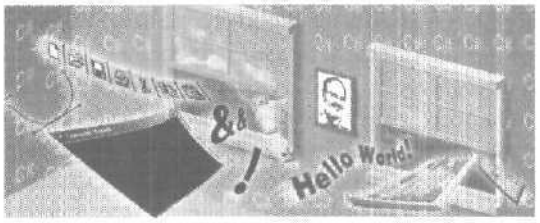
            atrackbar[i].Height = cy - 4 * cyFont;
            atrackbar[i].Location =
                new Point((1 + 2 * i) * cx / 12 - atrackbar[i].Width / 2,
                          2 * cyFont);

            alabelValue[i].Location = new Point(i * cx / 6,
                                                cy - 3 * cyFont / 2);
            alabelValue[i].Size = new Size(cx / 6, cyFont);
        }
    }
    void TrackBarOnValueChanged(object oSj, EventArgs ea)
    {
        for (int 1 = O; i < 3; i++)
            if(;TrackBar) obj ==fl"trackbar[i];
                alabelValue[i].Text = atrackbar[i].Value.ToString();

        BackColor = Color.FromArgb(atrackbar[0].Value,
                                   atrackbar[1].Value,
                                   atrackbar[2].Value);
    }
}
```

Как видите, разница между двумя программами не очень-то велика. Большая часть различий между полосами прокрутки и регуляторами отражена в коде, устанавливающем свойства *TrackBar*. В программе ColorTrackBar не надобности устанавливать свойство *TabStop*, зато нужно устанавливать свойства *Orientation* и *TickFrequency*. Кроме того, программа пользуется преимуществами того, что цвет фона полос прокрутки можно менять, устанавливая свойство *BackColor* в красный, зеленый или голубой. Метод *OnResize* реализован немного иначе, так как я решил оставить ширину полос по умолчанию.





ГЛАВА

13

Кривые Безье и другие сплайны

Что такое сплайн[1]? Даже изданные сравнительно недавно словари английского языка определяют значение слова «spline» как «гибкая форма для рисования кривых, сделанная из дерева, твердой резины или металла».[2] Это определение вызывает в воображении образ инженера прошлых лет, который, согнувшись над чертежом, орудует чудным гибким инструментом в попытке соединить кривой разбросанные по бумаге точки. В современных словарях можно найти более точное определение слова «сплайн»: «заданная математической функцией плавная кривая, соединяющая ряд точек... см. также кривая Безье».[3]

Пьер Этьен Безье родился в Париже в 1910 г. в семье инженера. В 1930 г. он завершил высшее образование, получив степень в машиностроении, а на следующий год дополнил ее степенью в электротехнике. В 1933 г. он был принят на работу в автомобильную компанию Рено (Франция), в которой и проработал до 1975 г. В 50-е Безье отвечал за внедрение одних из первых сверлильных и фрезерных станков с ЧПУ (т. е. с числовым программным управлением; в те дни этот термин встречался еще редко).

С 1960 г. Безье в основном занимался программой UNISURF — одной из первых систем автоматизированного проектирования и производства, использовавшейся в Рено для интерактивного конструирования автомобильных деталей. Для

---

[1] Транслитерация английского слова *spline,* ближайший русский эквивалент — *лекало,* — *Прим. перев.*

[2] Определение взято из словаря American Heritage Dictionary of the English Language, 4th ed. (Boston: Houghton Mifflin, 2000).

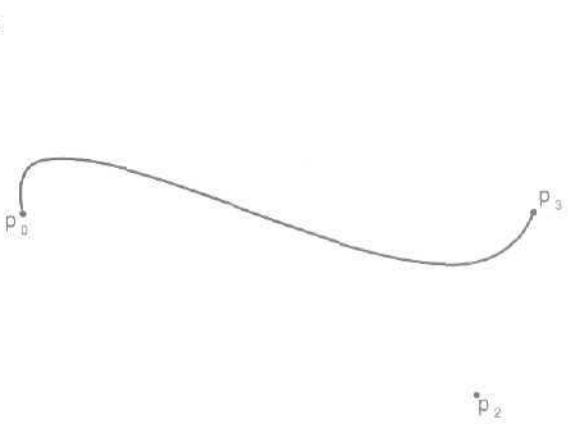
[3] Microsoft Computer Dictionary, 4th ed. (Redmond, WA: Microsoft Press, 1999),

UNISURF нужно было разработать пригодную к промышленному внедрению методику математического определения сложных кривых, которая позволила бы конструкторам манипулировать кривыми, ничего не зная о задающих их функциях. В результате этой самой работы и появились на свет кривые, которые теперь носят имя Безье. Пьер Безье скончался в 1999 г.[4]

Кривая Безье стала очень важным элементом компьютерной графики и занимает в табели о рангах третье место после прямой и эллиптической дуги. В языке PostScript *все* кривые визуализируются при помощи сплайнов Безье. Даже эллиптические дуги, и те аппроксимируют в PostScript кривыми Безье. Кривые Безье также используются для определения контуров шрифтов PostScript (в шрифтах TrueType применяется более простой и быстродействующий сплайн).

Знакомство с кривыми Безье на практике

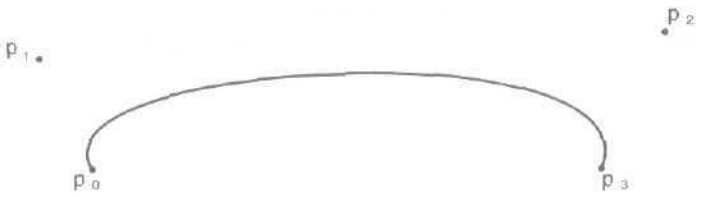
Кривая Безье уникально определяется четырьмя точками, которые можно обозначить p_0, p_1, p_2 и p_3. Кривая начинается в точке p_0 *и* кончается в точке p_3, поэтому p_0 обычно называют *начальной* точкой, p_3 — *конечной* (в совокупности их чисто начинают ***конечными)***. а p_1 и p_2 —***управляющими***. Управляющие точки действуют на кривую, как магниты. «притягивая» ее к себе. Вот пример кривой Безье:



Обратите внимание, что кривая, начавшись в точке p_0, направляется к p_1, но, не дойдя до нее, сворачивает и идет уже в сторону p_2. Не касаясь p_2, кривая продолжает свой путь и оканчивается в точке p_3. А вот еще одна кривая Безье:

---

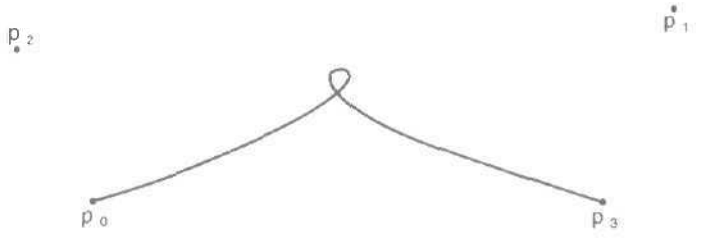
[4] Большая часть биографических сведений о П. Безье взята из его статьи «Style, Mathematics and NO. опубликованной в *22* томе журнала Computer-Aided Design, вышедшего в ноябре 1990 г. Две книги Безье были переведены на английский язык: Numerical Control: Mathematics and Application (London: John Wiley & Sons, 1972) и Pierre Bézier, The Mathematical Basis of the UNISURF CAD System (London: Butterworths, 1986). Также можно прочитать написанную им главу «How a Simple System Was Born» в книге Gerald Farm. Curves and Surfaces for Computer-Aided Geometric Design: A Practical Guide, 4th ed. (San Diego: Academic Press, 1997).



Кривая походит через две управляющие точки лишь в очень редких случаях. Например, если расположить обе управляющих точки на отрезке, соединяющем конечные точки, то кривая Безье превратится в прямую, проходящую через управляющие ТОЧКИ:



Возможна и другая крайность. Если расположить управляющие точки определенным образом, кривая Безье образует петлю:



Чтобы нарисовать кривую Безье в программе Windows Forms, нужно задать четыре точки структурами *Point, PointF* или восемью значения типа *float*:

Методы *DrawBezier* класса *Graphics*

```
void DrawBezier(Pen pen, Point pt0, Point pt1, Point pt2, Point pt3)
void DrawBezier(Pen pen, PointF ptf0, PointF ptf1, PointF ptf2,
                         PointF ptf3)
void DrawBezier(Pen pen, float x0, float y0, float x1, float y1,
                         float x2, float y2, float x3, float y3)
```

Иногда удобнее задать эти четыре точки в виде массива структур *Point* или *PointF* — этот прием поддерживают методы *DrawBeziers* (обратите внимание на множественное число в названии метода). Можно передать методу *DrawBeziers* массив из четырех структур *Point* или *PointF*. Этот метод также позволяет нарисовать несколько соединенных кривых Безье:

Методы *DrawBeziers* класса *Graphics*

```
void DrawBeziers(Pen pen, Point[] apt)
void DrawBeziers(Pen pen, PointF[] aptf)
```

При рисовании нескольких соединенных кривых Безье конечная точка одной кривой одновременно является начальной точкой следующей. Это значит, что для рисования каждой дополнительной кривой достаточно задать три новых точки.

Чтобы нарисовать N кривых Безье, в массиве должно быть $3N + 1$ элементов. Если число элементов массива не равно $3N + 1$, то при $N \geq 1$ метод генерирует исключение.

Методов *FillBezier* и *FillBeziers* не существует. Чтобы залить область, ограниченную кривыми Безье, применяются графические контуры, о которых пойдет речь в главе 15.

Чтобы почувствовать поведение кривых Безье, поэкспериментируйте с такой программой:

Bezier.cs

```
//-----------------------------------------
// Bezier.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class Bezier : Form
{
     protected Point[] apt = new Point[4];

     public static void Main()
     {
          Application.Run(new Bezier());
     }
     public Bezier()
     {
          Text = "Bezier (Mouse Defines Control Points)";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw = true;

          OnResize(EventArgs.Empty);
     }
     protected override void OnResize(EventArgs ea)
     {
          base.OnResize(ea);

          int cx = ClientSize.Width;
          int cy = ClientSize.Height;

          apt[0] = new Point(     cx / 4,      cy / 2);
          apt[1] = new Point(     cx / 2,      cy / 4);
          apt[2] = new Point(     cx / 2,  3 * cy / 4);
          apt[3] = new Point(3 * cx / 4,      cy / 2);
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          Point pt;
```

см. след. стр.

```
            if (mea.Button == MouseButtons.Left)
                pt = apt[1];

            else if (mea.Button == MouseButtons.Right)
                pt = apt[2];

            else
                return;

            Cursor.Position = PointToScreen(pt);
        }
        protected override void OnMouseMove(MouseEventArgs mea)
        {
            if (mea.Button == MouseButtons.Left)
            {
                apt[1] = new Point(mea.X, mea.Y);
                Invalidate();
            }
            else if (mea.Button == MouseButtons.Right)
            {
                apt[2] = new Point(mea.X, mea.Y);
                Invalidate();
            }
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics grfx = pea.Graphics;

            grfx.DrawBeziers(new Pen(ForeColor), apt);

            Pen pen = new Pen(Color.FromArgb(0x80, ForeColor));

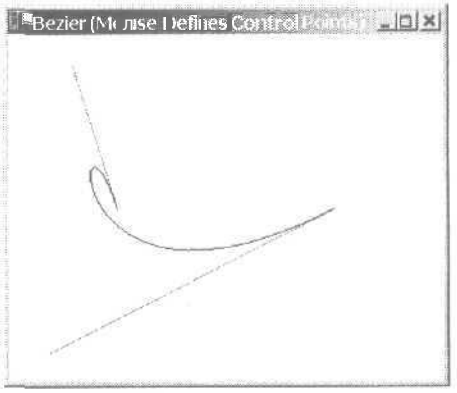
            grfx.DrawLine(pen, apt[0], apt[1]);
            grfx.DrawLine(pen, apt[2], apt[3]);
        }
    }
```

Программа фиксирует конечные точки кривой, но позволяет перемещать мышью управляющие точки. Точка p_1 перемещается левой кнопкой, а p_2 — правой. Функция «захвата» управляющих точек реализована здесь так: при нажатии левой или правой кнопки программа с помощью статического свойства *Cursor.Position* перемещает курсор к соответствующей управляющей точке. Программа также соединяет серыми линиями конечные точки с управляющими. Вот ее типичный вид (см. рис. на следующей странице).

Кривые Безье удобно использовать в программах художественной графики, так как они обладают рядом полезных свойств. Во-первых, после небольшой практики вы без труда сможете придать кривой желаемую форму.

Во-вторых, кривой Безье очень удобно управлять. Некоторые сплайны не проходят ни через одну из задающих их точек, кривая же Безье всегда закреплена двумя конечными точками (ниже мы увидим, что в этом состоит одна из предпосылок,

на которых строится вывод формул Безье). Кроме того, у некоторых сплайнов есть особые точки, в которых кривая разворачивается в бесконечность (что редко бывает желательно в компьютерной графике). Кривая Безье ведет себя намного лучше. Фактически она никогда не выходит за пределы четырехугольника (называемого *выпуклой оболочкой),* образованного линиями, соединяющими конечные и управляющие точки (способ соединения конечных и управляющих точек при образовании этого четырехугольника зависит от формы конкретной кривой).



Третье свойство кривых Безье касается связи между конечными и управляющими точками. Если провести из конечной точки луч через первую управляющую точку, то он всегда будет проходить по касательной к кривой Безье в ее конечной точке и будет направлен в ту же сторону, что и кривая. Это последние предпосылки для вывода формул Безье.

В-четвертых, кривые Безье часто принимают эстетичные формы, на которые приятно смотреть. Понимаю, что этот критерий субъективен, но я не единственный, кто находит кривые Безье весьма изящными.

Стильные часы

За два десятилетия, прошедших с момента появления программ, имитирующих аналоговые часы, их внешний вид почти не изменился. Почти всегда программисты рисуют стрелки часов в виде незатейливых прямоугольников, но теперь пришла пора осваивать новые возможности. Для этого мы нарисуем стрелки при помощи кривых Безье.

Вспомните программу AnalogClock из главы 10, использующую элемент управления из класса *ClockControl.* К счастью, я предусмотрительно изолировал код для рисования стрелок в защищенных виртуальных методах этого класса. Вот класс *BezierClockControl* с новыми методами *DrawHourHand* и *DrawMinuteHand,* вызывающими *DrawBeziers.*

BezierClockControl.cs

```
//-------------------------------------------------
// BezierClockControl.cs fl 2001 by Charles Petzold
//--
```

см. след. стр.

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

namespace Petzold.ProgrammingWindowsWithCSharp
{
class BezierClockControl: ClockControl
{
     protected override void DrawHourHand(Graphics grfx, Pen pen)
     {
          GraphicsState gs * grfx.Save();
          grfx.RotateTransform(360f * Time.Hour / 12 +
                               30f * Time.Minute / 60);

          grfx.DrawBeziers(pen, new Point[]
          {
          ;     new Point( 0, -600), new Point( 0, -300),
                new Point(200, -300), newPointC 50, -200),
                new PoiRt( 50, -200), new Point( 50, 0),
                new Point( 50, 0), new Point( 50, 75),
                new Point(-50, 75), new Point( -50, 0),
                new Point(-50, 0), new PointC -50, -200),
                new Point(-50, -200), new Point(-200, -300),
                new Point( 0, -300), new Point( 0, -600)
          });
          grfx.Restore(gs);
     }
     protected override void DrawMinuteHand(Graphics grfx, Pen pen)
     {
          GraphicsState gs = grfx.Save();
          grfx.RotateTransform(360f * Time.Minute / 80 '+ ,
                               6f * Time.Second / 80);

          grfx.DrawBeziers(pen, new Point[]
          {
               new Point( 0, -800), new Point( 0, -750),
               new Point( 0, -708), new Point( 25, -800),
               new Point( 25, -600), new Point( 25, 9),
               new Point( 25, 0), new Point( 25, 59),
               new Point(-25, 50), new Point( -25, 0),
               new Point(-25, 0), new PointC -25, -600),
               new Point(-25, -600), new Point( 0, -700),
               new Point( 0. -750), newPoint ( 0, -800)
          });
          grfx.Restore(gs);
     }
}
}
```

При каждом вызове методу *DrawBeziers* передается массив из 16 структур *PcInt* для рисования 5 кривых Безье (помните, что при использовании метода *DrawBeziers* для рисования первой кривой Безье требуется задать 4 точки, а для каждой дополнительной кривой — по 3).

Исходная программа AnalogClock так мала, что я не вижу смысла использовать ее в качестве родительского класса. Вместо этого я написал совершенно новую программу BezierClock на основе класса *BezierClockControl*.

BezierClock.cs

```
//---------------------------------------
// BezierClock.cs © 2001 by Charles Petzold
//---------------------------------------
using Petzold.ProgrammingWindowsWithCSharp;
using System;
using System.Drawing;
using System.Windows.Forms;

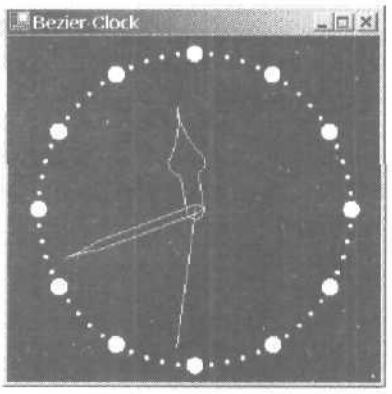
class BezierClock : Form
{
    BezierClockControl clkctl;

    public static void Main()
    {
        Application.Run(new BezierClock());
    }
    public BezierClock()
    {
        Text = "Bezier Clock";

        clkctl = new BezierClockControl();
        clkctl.Parent    = this;
        clkctl.Time      = DateTime.Now;
        clkctl.Dock      = DockStyle.Fill;
        clkctl.BackColor = Color.Black;
        clkctl.ForeColor = Color.White;

        Timer timer = new Timer();
        timer.Interval = 100;
        timer.Tick += new EventHandler(OnTimerTick);
        timer.Start();
    }
    void OnTimerTick(object obj, EventArgs ea)
    {
        clkctl.Time = DateTime.Now;
    }
}
```

А вот результат ее работы:



Фигурный кончик каждой стрелки рисуется двумя симметричными кривыми Безье. Прямая часть стрелки нарисована другой парой кривых Безье, а закругленная часть в центре циферблата — еще одной кривой. Таким образом, каждая стрелка нарисована пятью кривыми.

Коллинеарные точки и кривые Безье

Хотя у соединенных кривых Безье общие конечные точки, возможны ситуации, в которых переход одной кривой в другую не будет плавным. Говоря математическим языком, составная кривая считается плавной, только если ее первая производная непрерывна, т. е. не меняет резко направление.

При рисовании нескольких кривых Безье, соединенных вместе, часто желательно получить в результате плавную составную кривую, т. е. чтобы один сегмент кривой плавно переходил в другой. Но такая задача стоит не всегда. Например, стрелки часов — это совокупность плавных и прерывистых переходов. В точке встречи двух кривых Безье, рисующих кончик стрелки, первая производная результирующей кривой будет разорвана. Аналогичный разрыв будет и в точке соединения прямой части стрелки с фигурным контуром ее кончика. Но прямые, которыми нарисованы стрелки, плавно переходят в закругленную линию в центре циферблата.

Если нужен плавный переход между соединенными кривыми Безье, следующие три точки должны быть коллинеарными (лежать на одной прямой):

- вторая управляющая точка первой кривой;
- конечная точка первой кривой (она же начальная точка второй кривой);
- первая управляющая точка второй кривой.

Вот программа, рисующая четыре кривых Безье, соединенных плавными переходами. Конец четвертой кривой соединяется с началом первой, образуя замкнутый криволинейный контур.

Infinity.cs

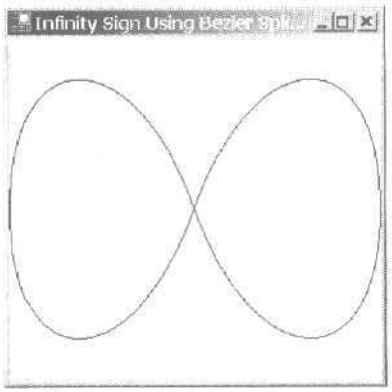
```
//---------------------------------------
// Infinity.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
```

см. след. стр.

```
using System.Drawing;
using System.Windows.Forms;

class Infinity: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new Infinity());
    }
    public Infinity()
    {
        Text = "Infinity Sign Using Bezier Splines";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        cx-;
        cy-;

        Point[] apt =
        {
            new Point(0,           cy / 2),     // Начальная
            new Point(0,           0),          // Управляющая
            new Point(    cx / 3, 0),          // Управляющая
            new Point(    cx / 2, cy / 2),     // Конечная / Начальная
            new Point(2 * cx / 3, cy),         // Управляющая
            new Point(    cx,     cy),         // Управляющая
            new Point(    cx,     ey / 2),     // Конечная / Начальная
            new Point(    cx,     0),          // Управляющая
            new Point(2 * cx / 3, 0),          // Управляющая
            new Point(    cx / 2, cy / 2),     // Конечная / Начальная
            new Point(    cx / 3, cy),         // Управляющая
            new Point(0,          cy),         // Управляющая
            new Point(0,          cy / 2)      // Конечная
        };
        grfx.DrawBeziers(new Pen(clr), apt);
    }
}
```

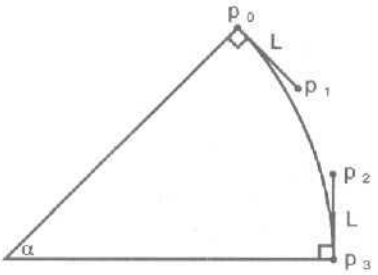


Каждая точка помечена в массиве как начальная, конечная/начальная или управляющая, любая точка коллинеарна двум соседним точкам. В результате эти четыре кривые образуют фигуру, похожую на символ бесконечности (см. предыдущий рисунок).

Рисование окружностей и дуг кривыми Безье

Выше я сказал, что в языке PostScript кривые Безье применяются для рисования эллиптических дуг. Как станет ясно в 15 главе, Windows Forms поступают аналогично, по крайней мере при сохранении дуг и эллипсов в виде графических контуров,

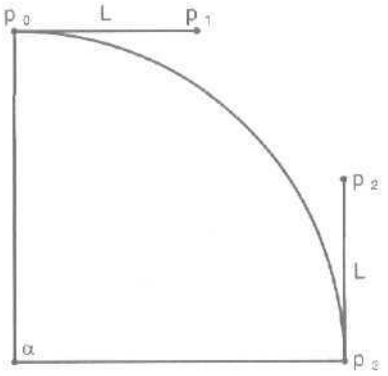
В литературе можно найти пару статей, посвященных аппроксимации эллиптических дуг кривыми Безье.[5] В первой описана довольно простая методика рисования круговых сегментов. Допустим, при помощи кривой Безье нужно нарисовать дугу окружности с заданным радиусом и угловой шириной a. Ясно, что p_0 и p_3 будут точками начала и конца дуги, но как задать p_1 и p_2? Как показывает эта диаграмма, проблема сводится к нахождению расстояния между конечными и управляющими точками, т. е. длины отрезка L:



Углы между отрезками, соединяющими конечные и управляющие точки, и радиусами сегмента обозначены на диаграмме как прямые. Откуда это следует? Из условия, коллинеарности, которое должно выполняться, чтобы кривая была плавной. Если вы собираетесь использовать кривую Безье, чтобы пририсовать к уже нарисованной дуге еще одну с центром в той же точке и с таким же радиусом, то общая конечная точка этих дуг и две соседние управляющие точки должны быть коллинеарными. Это значит, что отрезок, соединяющий конечную точку с управляющей, будет перпендикулярен радиусу окружности.

Если известна длина отрезка L, то для расчета координат p_1 и p_2 достаточно элементарной тригонометрии. Посмотрите, как просто рассчитать координаты точек p_1 и p_2, если радиусы сегмента находятся под прямым углом друг к другу и параллельны осям координат:

[5] Tor Dokkcn, et al., «Good Approximation of Circles by Curvature-Continuous Buzier Curves», Computer Aided Geometric Design 7 (1990), 33–41. Michael Goldapp, «Approximation of Circular Arcs by Cubic Polynomials», Computer Aided Geometric Design 8 (1991), 227-238.



В случае угла 180° вычисление координат p_1 и p_2 также тривиально.

Выкладки первой из процитированных мной статей доказывают, что получается неплохая аппроксимация, если взять для радиуса множитель:

$$L=\frac{4}{3}\tan\left(\frac{1}{4}\alpha\right)$$

При помощи этой аппроксимации программа BezierCircles рисует две полных окружности, первая из которых образована двумя кривыми Безье, а вторая (более точная) — четырьмя.

BezierCircles.cs

```
//---------------------------------------
// BezierCircles.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierCircles: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new BezierCircles());
     }
     public BezierCircles()
     {
          Text * "Bezier Circles";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          int iRadius = Math.Min(cx - 1, cy - 1) / 2;

          grfx.DrawEllipse(new Pen(clr), cx / 2 - iRadius, cy / 2 - iRadius,
                                         a * iRadius, 2 * iRadius);

               // Аппроксимация двумя сегментами по 180°
```

см. след. стр.

```
            int L = (int) Math.Round(iRadius * 4f / 3 * Math.Tan(Math.PI / 4));

            Point[] apt = {
                            new Point(cx / 2,      cy / 2 - iRadius),
                            new Point(cx / 2 + L,  cy / 2 - iRadius),
                            new Point(cx / 2 + L,  cy / 2 + iRadius),
                            new Point(cx / 2,      cy / 2 + iRadius),
                            new Point(cx / 2 - L,  cy / 2 + iRadius),
                            new Point(cx / 2 - L,  cy / 2 - iRadius),
                            new Point(cx / 2,      cy / 2 - iRadius)
                          };
            grfx.DrawBeziers(Pens.Blue, apt);

                // Аппроксимация четырьмя сегментами по 90°

            L = (int) Math.Round(iRadius * 4f / 3 * Math.Tan(Math.PI / 8));

            apt = new Point[]
                          {
                            new Point(cx / 2,      cy / 2 - iRadius),
                            new Point(cx / 2 + L,  cy / 2 - iRadius),
                            new Point(cx / 2 + iRadius, cy / 2 - L),
                            new Point(cx / 2 + iRadius, cy / 2),
                            new Point(cx / 2 + iRadius, cy / 2 + L),
                            new Point(cx / 2 + L,  cy / 2 + iRadius),
                            new Point(cx / 2,      cy / 2 + iRadius),
                            new Point(cx / 2 - L,  cy / 2 + iRadius),
                            new Point(cx / 2 - iRadius, cy / 2 + L),
                            new Point(cx / 2 - iRadius, cy / 2),
                            new Point(cx / 2 - iRadius, cy / 2 - L),
                            new Point(cx / 2 - L,  cy / 2 - iRadius),
                            new Point(cx / 2,      cy / 2 - IRadius)
                          };
            grfx.DrawBeziers(Pens.Red, apt);
        }
    }
```

Эта программа наглядно демонстрирует отличие аппроксимации кривыми Безье от метода *DrawEllipse*. Программа начинает обработку метода *DoPage* с вызова *DrawEllipse* для рисования эллипса с черным контуром. Контур фигуры, полученной в результате аппроксимации двумя кривыми Безье, рисуется синим цветом, а контур фигуры, образованной четырьмя кривыми, — красным. Помните: аргументы тригонометрических функций класса *Math* выражаются в радианах, поэтому вместо того, чтобы поделить угол на 4, как в формуле для расчета *L, я* использовал выражение на основе константы *Math.PI*.

«Картины Безье»

Многие, включая самого Безье[6], использовали кривые Безье для создания интересных фигур и узоров. Все подобные шедевры называются «картинами Безье». В этом творчестве нет правил. Единственное: при создании этих рисунков часто используют *цикл for*. Вот пример:

BezierArt.cs

```
//-----------------------------------------------------------------------
// BezierArt.cs © 2001 by Charles Petzold
//-----------------------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierArt: PrintableForm
 {
     const int iNum = 100;

     public new static void Main()
     {
          Application.fiunfnewBezierArt());
     }
     public BezierArt()
     {
          Text = "Bezier Art";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen      pen  = new Pen(clr);
          PointF[] aptf = new PointF[4];

          for (int i * 8; i < iNum; i++)
          {
               double dAngle = 2 * i * Math.PI / iNum;

               aptf[0].X *     cx / 2 + cx /  2 * (float) Math.Cos(dAngle);
               aptf[0].Y = 5 * cy / 8 + cy / 16 * (float) Math.Sin(dAngle);

               aptf[1] = new PointF(cx / 2,     -cy);
               aptf[2] * new PointF(cx / 2, 2 * cy);

               dAngle += Math.PI;

               aptf[3].X = cx / 2 + cx /  4 * (float) Math.Cos(dAngle);
```

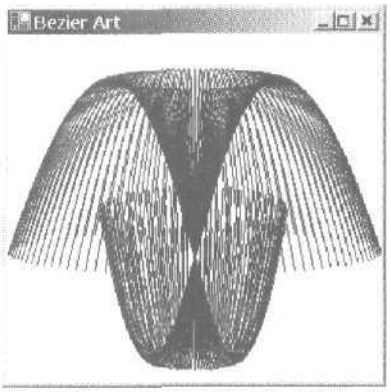
см, след. стр.

[6] Часть работ Пьера Безье можно увидеть на Web-узле профессора Брайана Барски по адресу http://www.cs.berkeley.edu/~barsky/gifs/bezier.html.

```
                aptf[3].Y * cy / 2 + cy / 16 * (float) Math.Sin(dAngle);

                grfx.DrawBeziers(pen, aptf);
            }
        }
    }
```

Фигуры, нарисованные множеством линий, обычно лучше выглядят на печати, но я рискну показать результат работы этой программы так, как он выглядит на экране:



Хотя в этой книге лишь черно-белые иллюстрации, что ограничивает мои возможности, не забывайте о цвете в своих картинах Безье.

Вывод формул Безье

Порой полезно знать формулы, по которым графическая система рассчитывает визуализацию кривых, например, чтобы расположить графические элементы (например, символы текста) по ходу кривой, нарисованной системой. Кроме того, *вывод* формул для расчета кривых — неплохое упражнение (не с Луны же они берутся!).

Кривая Безье — это полином третьего порядка. Как и все полиномы третьего порядка, кривая Безье уникально определяется четырьмя точками. Мы обозначили эти точки p_0 (начальная), p_1 и p_2 (две управляющие) и p_3 (конечная). Их можно обозначить и таю (x_0, y_0), (x_1, y_1), (x_2, y_2) и (x_3, y_3).

Полином третьего порядка, задающий координаты точек в двумерном пространстве, можно записать в форме параметрических уравнений общего вида:

$$x(t) = a_x \cdot t^3 + b_x \cdot t^2 + c_x \cdot t + d_x$$
$$y(t) = a_y \cdot t^3 + b_y \cdot t^2 + c_y \cdot t + d_y$$

где a_x, b_x, c_x, d_x, a_y, b_y, c_y и d_y — константы, а t меняется от 0 до 1. Любая кривая Безье уникально определяется этими 8 константами. Их значения зависят от координат четырех точек, задающих кривую. Цель этого упражнения — вывести уравнения для расчета 8 констант по заданным координатам 4 точек.

Наше первое допущение для вывода этих уравнений в том, что кривая Безье начинается в точке с координатами (x_0, y_0) при t=0:

$$x(0) = x_0$$
$$y(0) = y_0$$

Даже такое простое допущение позволяет продвинуться в выводе уравнений для констант. Подставив в параметрические уравнения $t=0$, получим:

$$x(0) = d_x$$
$$y(0) = d_y$$

Это означает, что две из констант — это просто координаты начальной точки:

$$d_x = x_0 \quad (1a)$$
$$d_y = y_0 \quad (1b)$$

Второе допущение, касающееся кривой Безье: она заканчивается в точке с координатами (x_3, y_3) при $t=1$:

$$x(1) = x_3$$
$$y(1) = y_3$$

Подставив в параметрические уравнения 1 вместо *t*, получаем:

$$x(1) = a_x + b_x + c_x + d_x$$
$$y(1) = a_y + b_y + c_y + d_y$$

что означает наличие следующей связи между константами и координатами конечной точки:

$$a_x + b_x + c_x + d_x = x_3 \quad (2a)$$
$$a_y + b_y + c_y + d_y = y_3 \quad (2b)$$

Остальные допущения касаются первых производных параметрических уравнений, описывающих угол наклона кривой. Первую производную параметрического уравнения общего вида, задающего полином третьего порядка как функцию переменной *t*, можно записать так:

$$x'(t) = 3a_x t^2 + 2b_x t + c_x$$
$$y'(t) = 3a_y t^2 + 2b_y t + c_y$$

Нас, в частности, интересует угол наклона кривой в конечных точках. Как известно, прямая, проведенная из начальной точки через первую управляющую точку, проходит по касательной к кривой Безье и направлена в ту же сторону, что и кривая. Обычно эту прямую задают параметрическими уравнениями:

$$x(t) = (x_1 - x_0)\ t + x_0$$
$$y(t) = (y_1 - y_0)\ t + y_0$$

где *t* изменяется от 0 до 1. Но можно задать ее и иначе:

$$x(t) = 3\ (x_1 - x_0)\ t + x_0$$
$$y(t) = 3\ (y_1 - y_0)\ t + y_0$$

где *t* изменяется от 0 до 1/3. Почему именно 1/3? Дело в том, что длина той части кривой Безье, по касательной к которой проходит прямая, проведенная из точки p_0 через p_1 и направленная в ту же сторону, что и кривая, равна примерно 1/3 «т общей длины кривой. Первые производные модифицированных параметрических уравнений можно записать так:

$x'(t) = 3\ (x_1 - x_0)$
$y'(t) = 3\ (y_1 - y_0)$

Если нужно рассчитать по этим уравнениям угол наклона кривой Безье при t=0, то:

$x'(0) = 3\ (x_1 - x_0)$
$y'(0) = 3\ (y_1 - y_0)$

Подставив t=0 в уравнение первой производной полинома третьего порядка, получим:

$x'(0) = c_x$
$y'(0) = c_y$

Это позволяет записать равенство:

$c_x = 3\ (x_1 - x_0)$ (3a)
$c_y = 3\ (y_1 - y_0)$ (3b)

Последнее допущение таково: прямая, проведенная из второй управляющей точки через конечную, является касательной к кривой Безье в ее конечной точке и направлена в ту же сторону, что и кривая. Иначе говоря:

$x'(1) = 3\ (x_3 - x_2)$
$y'(1) = 3\ (y_3 - y_2)$

Из уравнений общего вида следует, что:

$x'(1) = 3a_x + 2b_x + c_x$
$y'(1) = 3a_y + 2ft_y + c_y$

поэтому:

$3a_x + 2b_x + c_x = 3\ (x_3 - x_2)$ (4a)
$3a_y + 2b_y + c_y = 3\ (y_3 - y_2)$ (4b)

Выражения 1a, 2a, 3a и 4a дают четыре уравнения с четырьмя неизвестными, которые можно решить относительно a_x, b_x, c_x и d_x, выразив их через x_0, x_1, x_2 и x_3. Выполнив ряд алгебраических преобразований, получаем:

$a_x = -x_0 + 3x_1 - 3x_2 + x_3$
$b_x = 3x_0 - 6x_1 + 3x_2$
$c_x = 3x_0 + 3x_1$
$d_x = x_0$

Уравнения 1b, 2b, 3b и 4b позволяют сделать то же самое для коэффициентов y. После этого можно подставить константы обратно в параметрическое уравнение общего вида для полинома третьего порядка:

$x(t) = (?x_0 + 3x_1 - 3x_2 + x_3)\ t^3 + (3x_0 - 6x_1 + 3x_2)\ t^2 + (3x_0 + 3x_1)\ t + x_0$
$y(t) = (?y_0 + 3y_1 - 3y_2 + y_3)\ t^3 + (3y_0 - 6y_1 + 3y_2)\ t^2 + (3y_0 + 3y_1)\ t + y_0$

В сущности, на этом можно было бы закончить. Но лучше раскрыть скобки и привести подобные слагаемые. В итоге получатся более элегантные параметрические уравнения, с которыми проще работать:

$x(t) = (1 - t)^3 x_0 + 3t\ (1 - t)^2 x_1 + 3t^2\ (1 - t)\ x_2 + t^3 x_3$
$y(t) = (1 - t)^3 y_0 + 3t\ (1 - t)^2 y + 3t^2\ (1 - f)\ y + t^3 y_3$

Такими уравнениями обычно задают кривые Безье.

В следующей программе класс *BezierManual* переопределяет класс ***Bezier*** из программы Bezier, показанной выше, и рисует вторую кривую Безье. Но на эт:>т раз кривая рассчитывается «вручную» при помощи параметрических уравнений, которые мы только что вывели,

BezierManual.cs

```
//-------------------------------------------
// BezierManual.cs © 2001 by Charles Petzold
//-------------------------------------------
usingSystem;
using System.Drawing;
using System.Windows.Forms;

class BezierManual: Bezier
{
     public new static void Main()
     {
          Application.Run(new BezierManual());
     }
     public BezierManual()
     {
          Text - "Bezier Curve \"Manually\" Drawn";
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          base.OnPaint(pea);

          BezierSpline(pea.Graphics, Pens.Red, apt);
     }
     void BezierSpline(Graphics grfx, Pen pen, Point[] aptDefine)
     {
          Point[] apt = new Point[100];

        , for (int 1 = 0; i < apt.Length; i++)
          {
               float t = (float) i / (apt.Length - 1);

               float x = <1 - t) * (1 - t) * <1 - t) * aptDefine[0].X +
                         3 * t  * (1 - t) * (1 - t) * aptDefine[1].X +
                         3 * t * t       * (1 - t) * aptDefine[2].X *
                         t * t * t                 * aptDefine[3].X;

               float y = (1 - t) * (1 - t) * £1 - t) * aptDefine[0].Y +
                         3 * t  * (1 - t) * (1 - t)  * aptDefine[1].Y +
                         3 * t * t        * (1 - t)  * aptDefine[2].Y +
                         t * t * t                   * aptDefine[3].Y;

               apt[i] = new PointC(int)  Math.Round(x),  (int)  Math.Round(y));
```

см. след. стр.

```
            }
            grfx.DrawLines(pen,  apt);
        }
    }
```

Метод *OnPaint* в программе BezierManual вызывает одноименный метод из базового класса (которым является класс *Bezier*), а затем — метод *BezierSpline* из своего класса. Определение метода *BezierSpline* не отличается от *DrawBeziers* за исключением использования объекта *Graphics* в качестве первого аргумента и возможности обработки лишь одной кривой Безье. Этот метод задействует массив из 100 структур *Point,* каждая из которых рассчитывается на основе полученных выше параметрических уравнений, после чего рисует кривую как составную линию. Программа рисует кривую Безье, рассчитанную «вручную», красным цветом, что позволяет сравнить ее с аналогичной кривой, нарисованной Windows Forms. Кривые, конечно, отличаются, но не более чем на 1 пиксел.

Канонический сплайн

В классе *Graphics* есть еще один тип сплайна — *канонический (стандартный* или *обычный).* Его рисуют методы *DrawCurve.* Существует в общей сложности семь версий этого метода, но чаще используются четыре:

Методы *DrawCurve* класса *Graphics* (выборочно)

```
DrawCurve(Pen pen, Point[] apt)
DrawCurve(Pen pen, PointF[] aptf)
DrawCurve(Pen pen, Point[] apt, float fTension)
DrawCurve(Pen pen, PointF[] aptf, float fTension)
```

Для этого метода нужно задать минимум две точки. Если массив, переданный методу, содержит всего две точки, то *DrawCurve* соединяет их прямой. Если точек три или больше, метод рисует кривую, соединяющую все точки.

Существенное отличие канонического сплайна от кривых Безье в том, что он проходит через все точки, заданные массивом. Иногда участок кривой между двумя соседними точками называют *сегментом* кривой. Форма каждого сегмента определяется не только начальной и конечной точками сегмента, но и их соседними точками. Допустим, кривая задана массивом структур *Point* с именем *apt.* В этом случае форма сегмента, ограниченного точками *apt[3]* и *apt[4],* также будет определяться точками *apt[2]* и *apt[5].*

Форма сплайна, помимо прочего, зависит от *натяжения* (tension) — параметра, который задают как явный аргумент в перегруженных версиях метода *DrawCurve.* По своей сути этот параметр напоминает жесткость традиционного гибкого лекала. По умолчанию натяжение равняется 0,5. Натяжение, равное 0, дает прямые, т. е. метод *DrawCurve* превращается в *DrawLines.* При натяжении больше 0,5 изгиб кривой увеличивается. Если задать натяжение меньше 0 (или больше 1), кривая может сделать петлю.

Поэкспериментируем? Следующая программа очень похожа на Bezier, но дополнена полосой прокрутки, позволяющей определить натяжение, и дает больше свободы при перемещении точек,

CanonicalSpiine.cs

```
//-----------------------------------------------
// CanonicalSpline.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CanonicalSpline: Form
{
     protected Point[] apt      = new Point[4];
     protected float   fTension = 0.5f;

     public static void Main()
     {
          Application.Run(new CanonicalSpline());
     }
     public CanonicalSpline()
     {
          Text = "Canonical Spline";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw - true;

          ScrollBar scroll - new VScrollBar();
          scroll.Parent - this;
          scroll.Dock = DockStyle.Right;
          scroll.Minimum - -100;
          scroll.Maximum = 108;
          scroll.SmallChange =  1;
          scroll.LargeChange = 10;
          scroll.Value = (int) (10 * fTension);
          scroll.ValueChanged += new EventHandler(ScrollOnValueChanged);

          OnResize(EventArgs.Empty);
     }
     void ScrollOnValueChanged(object obj, EventArgs ea)
     {
          ScrollBar scroll = (ScrollBar) obj;

          fTension = scroll.Value / 1Sf;

        • Invalidate(false);
     }
     protected override void OnResize( Event Args ea)
     {
          base.OnResize(ea);

          int GX = ClientSize.Width;
```

см. след.стр.

```
        int cy * ClientSize.Height;

        apt[0] = new Point(    cx / 4,       cy / 2);
        apt[1] = new Point(    GX / 2,      cy / 4);
        apt[2] = newPoint(     cx /2,  3 * cy /  4);
        apt[3] = new Point(3 * cx /4,       cy / 2);
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
        Point pt;

        if (mea.Button == MouseButtons.Left)
        Г
            if (ModifierKeys —— Keys.Shift)
                pt - apt[0];
            else if (ModifierKeys == Keys.None)
                pt = apt[1];
            else
                return;
        }
        else if (mea.Button== MouseButtons.Right)
        {
            if (ModifierKeys == Keys.None)
                pt = apt[2];
            else if (ModifierKeys -- Keys.Shift)
                •pt = apt[3];
            else
                return;
        }
        else
            return;

        Cursor.Position = PointToScreen(pt);
    >
    protected override void OnMouseMove(MouseEventArgs mea)
    {
        Point pt = new Point(mea.X, mea.Y);

        if (mea.Button == MouseButtons.Left)
        {
            if (ModifierKeys == Keys.Shift)
                apt[0] = pt;
            else if (ModifierKeys —— Keys.None)
                apt[1] = Pt;
            else
                return;
        I
        else if (mea.Button == MouseButtons.flight)
        {
```

см. след. стр.

```
            if (ModifierKeys == Keys.None)
                apt[2]= pt;
            else if (ModifierKeys == Keys.Shift)
                apt[3] = pt;
            else
                return;
        }
        else
            return;

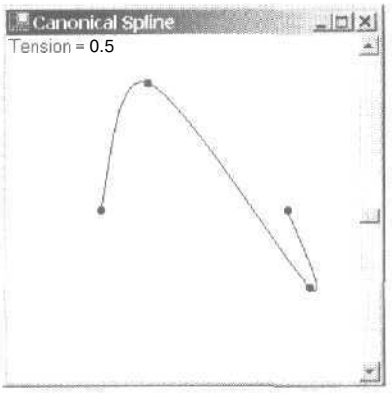
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx  = pea.Graphics;
        Brush    brush = new SolidBrush(ForeColor);

        grfx.DrawCurve(new Pen(ForeColor), apt, fTension);

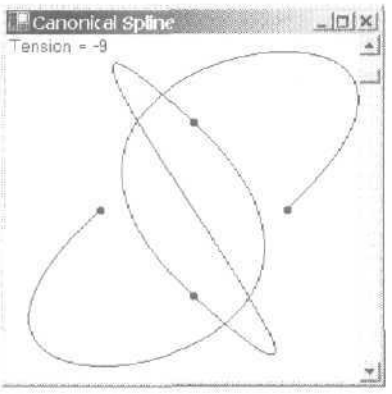
        grfx.DrawString("Tension = " + fTension, Font, brush, 0, 0);

        for (int i = 0; I < 4; i++)
            grfx.FillEllipse(brush, apt[i].X - 3, apt[i].Y - 3, 7, 7);
    }
}
```

Как и в программе Bezier, левая и правая кнопки мыши служат для переме щения точек p_1 и p_2. Кроме того, при нажатии клавиши Shift программа CanonicalSpline позволяет перемещать точки p_0 и p_3 левой и правой кнопками. Обычно эта программа выглядит так:



Натяжение регулируется положением ползунка на полосе прокрутки, значение натяжения выводится в левом верхнем углу окна программы. Я разрешил менять натяжение от 10 до -10, чтобы стало видно, как крайние значения заставляют кривую выделывать сумасшедшие петли. Вот одна из моих любимых кривых, созданная при помощи заданного по умолчанию массива структур *Point:*



Следующие методы *DrawCurve* позволяют задействовать подмножество точек массива:

Методы *DrawCurve* класса *Graphics* (выборочно)

```
DrawCurve(Pen pen, PointF[] aptf, int iOffset, int iSegments)
DrawCurve(Pen pen, Point[] apt, int iOffset, int iSegments,
          float fTension)
DrawCurve(Pen pen, PointF[] aptf, int iOffset, int iSegments,
          float fTension)
```

Аргумент *iOffset* — это индекс начальной точки кривой в массиве *Point* или *PointF*. Аргумент *iSegments* задает число рисуемых сегментов, а кроме того, показывает, сколько дополнительных структур *Point* или *PointF* будет использовать метод. Вызов:

```
grfx.DrawCurve(pen, aptf, 2, 3);
```

рисует три сегмента: *aptf[2]–aptf[3]*, *aptf[3]–aptf[4]* и *aptf[4]–aptf[5]*. Результат выглядит так же, как вызов более простой версии *DrawCurve* с теми же четырьмя точками, заданными явно. В версии *DrawCurve*, использующей аргументы *iOffset* форму сегмента *aptf[4]–aptf[5]*.

Метод *DrawClosedCurve* соединяет первую и последнюю точки массива дополнительной кривой:

Методы *DrawClosedCurve* класса *Graphics*

```
DrawClosedCurve(Pen pen, Point[] apt)
DrawClosedCurve(Pen pen, PointF[] aptf)
DrawClosedCurve(Pen pen, Point[] apt, float fTension, FillMode fm)
DrawClosedCurve(Pen pen, PointF[] aptf, float fTension, FillMode fm)
```

Своеобразие *DrawClosedCurve* не только в том, что он рисует дополнительный сегмент. Первый сегмент, нарисованный этим методом, несколько отличается от сегмента, нарисованного *DrawCurve*, поскольку на его форму также влияет последняя точка массива. Аналогично первая точка массива влияет на форму предпоследнего сегмента.

У двух перегруженных версий метода *DrawClosedCurve* есть аргумент *FillMode*. Вы, конечно, помните перечисление *FillMode*, определяемое в пространстве имен *System.Drawing.Drawing2D*. Оно используется методом *DrawPolygon* и определяет, какие замкнутые области будут залиты, а какие — нет:

Перечисление *FillMode*

Член	Значение	Примечание
Alternate	0	Задано по умолчанию, чередует залитые и свободные области
Winding	1	Закрашивает большинство замкнутых областей

Но зачем, спросите вы, указывать режим заливки в методе, который просто рисует линии и ничего не закрашивает? Эта тайна покрыта мраком. Похоже, метод работает совершенно независимо от значения *FillMode*.

Намного больше смысла у этого аргумента в методах *FillClosedCurve*:

Методы *FillClosedCurve* класса *Graphics*

```
FillClosedCurve(Brush brush, Point[] apt)
FillClosedCurve(Brush brush, PointF[] aptf)
FillClosedCurve(Brush brush, Point[] apt, FillMode fm)
FillClosedCurve(Brush brush, PointF[] aptf, FillMode fm)
FillClosedCurve(Brush brush, Point[] apt, FillMode fm, float fTension)
FillClosedCurve(Brush brush, PointF[] aptf, FillMode fm, float fTension)
```

Программа ClosedCurveFillModes почти идентична FillModesClassical из главы 5. Она рисует две пятиконечные звезды, чтобы продемонстрировать отличие между методами *FillMode.Alternate* и *FillMode.Winding*:

ClosedCurveFillModes.cs

```
//-----------------------------------------
// ClosedCurveFillModes.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ClosedCurveFillModes: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new ClosedCurveFillModes());
     }
     ClosedCurveFillModes()
     {
          Text = "FillClosedCurve Fill Modes";
          ClientSize = new Size(2 * ClientSize.Height, ClientSize.Height);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
```

см. след. стр.

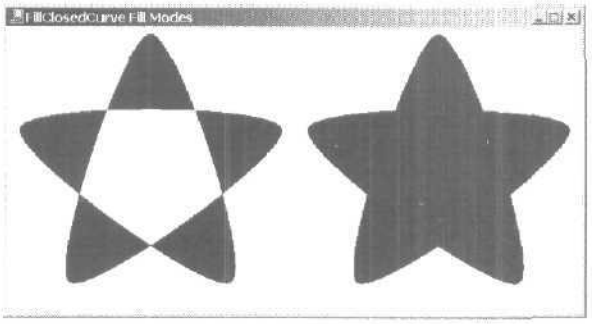
```
    {
        Brush  brush = new SolidBrush(clr);
        Point[] apt  = new Point[5];

        for (int i = 0; i < apt.Length; i++)
        {
            double dAngle = (i * 0.8 - 0.5) * Math.PI;
            apt[i] = new Point(
                        (int)(cx *(0.25 + 0.24 * Math.Cos(dAngle))),
                        (iRt)(cy *(0.50 + 0.48 * Math.Sin(dAngle))));
        }
        grfx.FillClosedCurve(brush, apt, FillMode.Alternate);

        for (int i * 0; i < apt.Length; i++)
            apt[i].X += cx / 2;

        grfx.FillClosedCurve(brush, apt, FillMode.Winding);
    }
}
```

Хотя в этих фигурах нетрудно узнать звезды, у них закругленные лучи:



Они больше похожи на печенья в форме звездочек. Когда те выходят из-под формы, их лучи еще остры, но в духовке тесто немного поднимается, и печенья становятся округлыми, совсем как наши фигуры.

Вывод уравнений канонической кривой

Подобно кривой Безье канонический сплайн — это полином третьего порядка, который можно записать в виде параметрических уравнений общего вида:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$
$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

где t изменяется от 0 до 1. Первые производные этих уравнений будут таковы;

$$x'(t) = 3a_x t^2 + 2b_x t + c_x$$
$$y'(t) = 3a_y t^2 + 2b_y t + c_y$$

Рассмотрим четыре точки, обозначенные p_0, p_1, p_2, и p_3. Я собираюсь вывести формулы для сегмента $p_1 - p_2$, форма которого определяется этой парой точек, а

также парой соседних точек, p_0 и p_3. Во-первых, допустим, что кривая начинается в точке p_1 и заканчивается в точке p_2:

$$x(0) = x_1$$
$$y(0) = y_1$$
$$x(1) = x_2$$
$$y(1) = y_2$$

Из параметрических функций общего вида можно вывести такие уравнения:

$$d_x = x_1$$
$$d_y = y_1$$
$$a_x + b_x + c_x + d_x = x_2$$
$$a_y + b_y + c_y + d_y = y_2$$

Другие допущения касаются угла наклона линии в точках p_1 и p_2. Допустим, угол наклона в точке p_1 является произведением натяжения (обозначим его через T) и угла наклона прямой, проведенной через точки p_0 и p_2. Аналогично предполагается, что угол наклона в точке p_2 равняется произведению натяжения на угол наклона прямой p_1p_3:

$$x'(0) = T(x_2 - x_0)$$
$$y'(0) = T(y_2 - y_0)$$
$$x'(1) = T(x_3 - x_1)$$
$$y'(1) = T(y_3 - y_1)$$

Из первых производных параметрических уравнений общего вида находим:

$$c_x = T(x_2 - x_0)$$
$$c_y = T(y_2 - y_0)$$
$$3a_x + 2b_x + c_x = T(x_3 - x_1)$$
$$3a_y + 2b_y + c_y = T(y_3 - y_1)$$

Выполнив алгебраическое преобразование и решив систему уравнений, получаем:

$$a_x = T(x_2 - x_0) + T(x_3 - x_1) + 2x_1 - 2x_2$$
$$a_y = T(y_2 - y_0) + T(y_3 - y_1) + 2y_1 - 2y_2$$
$$b_x = -2T(x_2 - x_0) - T(x_3 - x_1) - 3x_1 + 3x_2$$
$$b_y = -2T(y_2 - y_0) - T(y_3 - y_1) - 3y_1 + 3y_2$$
$$c_x = T(x_2 - x_0)$$
$$c_y = T(y_2 - y_0)$$
$$d_x = x_1$$
$$d_y = y_1$$

Программа CanonicalSplineManual позволяет убедиться, что мы не ошиблись при выводе этих констант.

CanonicalSplineManual.cs

```
//---------------------------------------------------------
// CanonicalSplineManual.cs © 2001 by Charles Petzold
//---------------------------------------------------------
using System;
using System.Drawing;
```

см. след. стр.

```
using System.Windows.Forms;

class CanonicalSplineManual: CanonicalSpline
{
     public new static void Main()
     {
          Application.Run(new CanonicalSplineManual());
     }
     public CanonicalSplineManual()
     {
          Text - "Canonical Spline \"Manually\" Drawn";
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          base.OnPaint(pea);

          CanonicalSpline(pea.Graphics, Pens.Red, apt, fTension);
     }
     void CanonicalSpline(Graphics grfx, Pen pen, Point[] apt, float T)
     f
          CanonicalSegment(grfx, pen, apt[0], apt[0], apt[1], apt[2], T);
          CanonicalSegment(grfx, pen, apt[0], apt[1], apt[2], apt[3], T);
          CanonicalSegment(grfx, pen, apt[1], apt[2], apt[3], apt[3], T);
     }
     void CanonicalSegment(Graphics grfx, Pen pen, Point pt0, Point pt1,
                           Point pt2, Point pt3, float T)
     {
          Point[] apt = new Point[10];

          float SX1 = T * <pt2.X - pt0.X);
          float SY1 = T * (pt2.Y - pt0.Y);
          float SX2 = T * (pt3.X - pt1.X);
          float SY2 = T * (pt3.Y - pt1.Y);
          float AX - SX1 + 8X2 + 2 * pt1.X - 2 * pt2.X;
          float AY - SY1 + SY2 + 2 * pt1.Y - 2 * pt2.Y;
          float BX * -2 * SX1 - SX2 - 3 * pt1.X + 3 * pt2.X;
          float BY = -2 * SY1 - SY2 - 3 « pt1.Y * 3 * pt2.Y;
          float CX * SX1;
          float CY = SY1;
          float DX = pt1.X;
          float DY = pt1.Y;

          for (int i - 0; 1 < apt.Length; i++)
          {
               float t = (float)i / (apt.Length - 1);
               apt[i].X - (int) <AX * t * t * t + BX * t * t + CX * t + DX);
               apt[i].Y = (int) (AY* t * t * t + BY * t * t * GY * t + DY);
          }
          grfx.DrawLines(pen, apt);
     }
}
```

Здесь я хочу обратить ваше внимание на пару моментов. Метод *CanonicalSpline* обрабатывает только массивы из четырех элементов и трижды вызывает *CanonicalSegment,* рисуя каждый раз по одному сегменту. С первым и последним сегментами нужно обращаться по-особому, поскольку кривая задана всего лишь тремя точками, а не четырьмя.

Для рисования каждого сегмента метод *CanonicalSegment* использует массив из 10 структур. Этого маловато для получения гладкой кривой, но вполне достаточно, чтобы продемонстрировать, что он действительно неплохо воспроизводит метод *DrawCurve,* реализованный в классе *Graphics.*

В главах 15 и 19 мы еще встретимся с программами, использующими кривые Безье и канонические сплайны.



ГЛАВА

14

Меню

Меню — важнейший элемент большинства традиционных приложений Microsoft Windows. Меню находится под заголовком формы и отражает практически все, что программа может делать: от простых операций типа выделения и вставки до сложных задач вроде проверки орфографии. Даже если в приложении предусмотрены разнообразные «быстрые клавиши», обычно их функции дублируются пунктами меню.

Меню большинства Windows-приложений выглядит очень похожими. И это важный аспект пользовательского интерфейса Windows. Пользователям легче привыкнуть к работе с новой программой, если меню программы похоже на меню других Windows-программ. При разработке меню имеет смысл обратить внимание на структуру и содержание меню существующих приложений. Я не хочу сказать, что следует увековечивать плохие решения, но иногда даже несовершенный интерфейс хорош просто потому, что к нему уже привыкли.

Visual Studio .NET содержит Menu Designer (Дизайнер меню), позволяющий интерактивно строить иерархическое меню. Довольно простой в использовании Menu Designer хорошо подходит для создания несложных меню. Но я почти уверен, что однажды вам не хватит Menu Designer и что вам не понравится генерируемый им код. Поэтому в данной главе я рассматриваю ручное кодирование меню.

Меню и его пункты

В Windows Forms меню, находящееся между заголовком формы и клиентской областью, называется *главным меню* формы. Кроме того, во многих приложениях используются *меню быстрого вызова*, или *контекстные меню*, появляющиеся под курсором при нажатии правой кнопки мыши. Главное меню связано с формой, а контекстное — обычно с конкретным элементом управления, т. е. щелчки разных элементов управления могут вызывать разные контекстные меню.

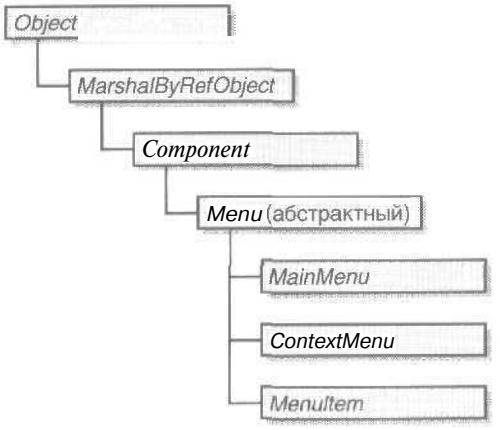
Любое меню (как главное, так и контекстное) содержит *пункты меню*. Пункт меню обычно связан со словом или короткой фразой: File (Файл), Open (Открыть). Save (Сохранить), Save As (Сохранить как), Edit (Правка), Cut (Вырезать), Copy (Копировать), Paste (Вставить) или Select All (Выделить все) и т. п. Далее я часто буду ссылаться на пункт меню по связанному с ним тексту.

Скорее всего вы слегка удивлены. Как же так, ведь пункты File и Edit явно отличаются от остальных перечисленных! Пункты File и Edit расположены в видимой части главного меню программы, а прочие нет. Пункты в видимой части главного меню называются пунктами *верхнего уровня*. Выбор пункта File или другого пункта верхнего уровня приводит к появлению прямоугольника, который раньше обычно называли *всплывающим* или *выпадающим* меню, но теперь чаще называют *подменю* или *дочерним* меню. (Именно так — еще один пример родительских отношений в Windows Forms!) Подменю, вызываемое из пункта File содержит дополнительные пункты Open, Save, Save As и т. д.

С точки зрения программы Windows Forms, пункт меню File содержит *массив* других пунктов меню, и этот массив включает элементы Open, Save и Save As. Пункт Edit содержит массив пунктов меню, включающий элементы Cut, Copy и Select All,

На самом деле, если подняться на уровень выше, станет ясно, что главное меню само по себе является массивом пунктов меню, включающим элементы File, Edit и все остальные из видимой части главного меню, причем часто последним пунктом является Help (Справка). Каждый пункт в главном меню связан с собственном массивом пунктов меню, и каждый массив является подменю главного меню. Некоторые из пунктов в них также содержат собственные массивы пунктов меню для вызова вложенных подменю. Точно так же контекстные меню являются массивом пунктов меню, каждый из которых может ссылаться на дополнительные массивы пунктов меню.

Классы *MainMenu*, *ContextMenu* и *MenuItem* наследуют абстрактному классу *Menu* из пространства имен *System.Windows.Forms*Далее приведена иерархия классов:



В первую очередь вы должны обратить внимание на то, что *Menu* не является наследником *Control* и такие знакомые свойства, как *BackColor*, *ForeColor* и *Font*, недоступны. Только пользователи могут изменять цвет и шрифт меню, но не программы. Если вы хотите выводить меню нестандартным шрифтом и в нестандарт-

ном цвете, вам придется задействовать возможность прорисовки владельцем, которую я опишу в конце этой главы,

Я упомянул, что главное меню формы является массивом пунктов меню. Вот два конструктора *MainMenu*, второй из которых демонстрирует этот факт:

Конструкторы *MainMenu*

```
MainMenu()
MainMenu(MenuItem[] ami)
```

Чтобы связать конкретный объект *MainMenu* с формой, надо присвоить его свойству *Menu* формы:

Свойства *Form* (выборочно)

Тип	Свойство	Доступ
MainMenu	*Menu*	Чтение/запись

Точно так же *ContextMenu* содержит два конструктора, второй из которых демонстрирует то, что контекстное меню является массивом элементов меню:

Конструкторы *ContextMenu*

```
ContextMenu()
ContextMenu(MenuItem[] ami)
```

Свойство *ContextMenu* класса *Control* позволяет связать конкретный объект *ContextMenu* с любым элементом управления:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
ContextMenu	*ContextMenu*	Чтение/запись

Если на форме несколько элементов управления или если форма разделена на несколько функциональных областей с помощью панелей, каждый элемент может вызывать собственное контекстное меню. Но при этом для реализации контекстных меню не обязательно использовать свойство элемента управления *ContextMenu*. Вместо этого можно вручную вызывать контекстные меню в методе *OnMouseUp* в зависимости от текущего местоположения курсора.

Хотя конструкторы *MainMenu* и *ContextMenu* указывают на то, что эти объекты связаны с массивами пунктов меню, я еще не ознакомил вас с конструкторами *MenuItem*. Вот первые пять из шести:

Конструкторы *MenuItem* (выборочно)

```
MenuItem()
MenuItem(string strText)
MenuItem(string strText, EventHandler ehClick)
MenuItem(string strText, EventHandler ehClick, Shortcut sc)
MenuItem(string strText, MenuItem[] ami)
```

MenuItem имеет еще один, довольно сложный конструктор, осуществляющий слияние меню.

Мы еще рассмотрим класс *MenuItem* гораздо подробнее, но будет невредно прямо сейчас привести несколько примеров. Для пункта меню верхнего уровня вроде File может применяться последний конструктор из таблицы:

```
new MenuItem("&File", amiFile)
```

Амперсант перед F в слове «File» указывает на то, что F должна быть подчеркнута и что она будет частью встроенного клавиатурного интерфейса Windows для меню. Когда пользователь нажимает клавишу Alt и затем F, выводится подменю для File, Для отображения амперсанта в тексте меню, его надо ввести дважды.

Параметром *amiFile* для только что приведенного конструктора будет массив пунктов меню, содержащий пункты Open, Save, Save As, и т. д. Один из элементов этого массива — пункт меню Open, который мог быть создан конструктором:

```
new MenuItem("&Open...",
             new EventHandler(MenuFileOpenOnClick),
             Shortcut.CtrlO)
```

И опять амперсант указывает на то, что буква «O» должна быть подчеркнута. Нажатие Alt, F и затем O вызывает пункт Open. Многоточие указывает на то, что пункт меню вызывает диалоговое окно.

Каждый пункт меню, *не вызывающий* подменю, обычно связан с обработчиком события *Click,* которое происходит, когда пользователь щелкает пункт меню или нажимает соответствующую клавиатурную комбинацию. Для таких обработчиков событий я буду использовать стандартную схему наименования. В реальных программах метод *MenuFileOpenOnClick* отвечает за вывод диалогового окна, позволяющего пользователю выбрать открываемый файл. (О диалоговых окнах см. главу 16.)

Быстрые клавиши меню

Я уже упоминал клавиатурный интерфейс меню, связанный с подчеркнутыми буквами. Когда пользователь нажимает клавишу Alt, форма переходит в режим выбора меню. Нажатие клавиши F выводит подменю File, а нажатие O эквивалентно щелчку пункта Open.

В Windows есть еще один интерфейс для работы меню, построенный на *акселераторах,* или *быстрых клавишах* в терминологии Windows Forms. Быструю клавишу указывают значением из перечисления *Shortcut.* В предыдущем примере значение *Shortcut.CtrlO* указывает на то, что сочетание клавиш Ctrl+O является быстрой клавишей для вывода диалогового окна открытия файла. Когда быстрые клавиши применяются в пунктах меню, текст «Ctrl+O» автоматически вставляется в видимый пользователю текст пункта меню. Когда пользователь нажимает Ct. 1+O, напрямую вызывается обработчик события *MenuFileOpenOnClick* без видимого участия меню.

Shortcut — это перечисление 150 клавиатурных сочетаний, рекомендованных для меню. (Это четвертое по размеру перечисление в .NET Framework.) В наборе, приведенном в первой таблице, приведены функциональные клавиши, а т юсе Insert, Delete и Backspace;

Перечисление *Shortcut* (выборочно)

F1	ShiftF1	CtrlFl	CtrlSbiftFl	AltF1
F2	ShiftF2	CtrlF2	CtrlShiftF2	AltF2
F3	ShiftF3	CtrlF3	CtrlShiftF3	AltF3
F4	ShiftF4	CtrlF4	CtrlShiftF4	AltF4
F5	ShiftF5	CtrlF5	CtrlShiftF5	AltF5
F6	ShiftF6	CtrlF6	CtrlShiftF6	AltF6
F7	ShiftF7	CtrlFl	CtrlSbiftFl	AltFl
F8	ShiftF8	CtrlF8	CtrlShiftF8	AltF8
F9	ShiftF9	CtrlF9	CtrlShiftF9	AltF9
F10	ShiftF10	CtrlF10	CtrlShiftF10	AlfF10
Fl 1	ShiftF1	CtrlFl 1	CtrlSbiftFl 1	AltFl 1
F12	ShiftF12	CtrlF12	CtrlShiftF12	AltF12
Ins	ShiftIns	Ctrllns		
Del	ShiftDel	CtrlDel		
None				AltBksp

Значение по умолчанию *ShortcutNone*. В следующей таблице приведены сочетания всех букв с клавишами Shift и Ctrl:

Перечисление *Shortcut* (выборочно)

CtrlA	CtrlN	CtrlShiftA	CtrlShiftN
CtrlB	CtrlO	CtrlShiftB	CtrlSbiftO
CMC	CMP	CtrlShiftC	CtrlShiftP
CtrlD	CtrlQ	CtrlShiftD	CtrlShiftQ
CtrlE	CtrlR	CtrlShiftE	CtrlShiftR
CtrlF	CtrlS	CtrlShiftF	CtrlShiftS
CtrlG	CtrlT	CtrlShiftG	CtrlShiftT
CtrlH	CtrlU	CtrlShiftH	CtrlShiftU
CtrlI	CtrlV	CtrlShiftI	CtrlShiftV
Ctrl]	CtrlW	CtrlShift]	CtrlShiftW
CtrlK	CtrlX	CtrlShiftK	CtrlShiftX
CtrlL	CtrlY	CtrlShiftl	CtrlShiftY
CtrlM	CtrlZ	CtrlShiftM	CtrlShiftZ

Значения этих членов перечисления эквивалентны значениям перечисления *Keys* (см. главу 6) из свойства *KeyData* класса *KevEventArgs*. Ниже перечислены сочетания числовых клавиш с клавишами Ctrl и Alt:

Перечисление *Shortcut* (выборочно)

Ctrl0	CtrlShift0	Alt0
Ctrll	CtrlShift1	Alt1
Ctrl2	CtrlShift2	Alt2
Ctrl3	CtrlShift3	Alt3
Ctrl4	CtrlShift4	Alt4
Ctrl5	CtrlShift5	Alt5

Перечисление *Shortcut* (выборочно) *(продолжение)*

Ctrl6	*CtrlShift6*	*Alt6*
Ctrl7	*CtrlShift7*	*Alt7*
Ctrl8	*CtrlShift8*	*Alt8*
Ctrl9	*CtrlShift9*	*Alt9*

Использовать в качестве быстрых клавиш меню значения, не определенные в перечислении *Shortcut*, нельзя.

В Win32-программировании клавиши-акселераторы можно использовать не только для быстрого вызова пунктов меню. Но в программе Windows Forms быстрые клавиши служат только для вызова пунктов меню. Быструю клавишу, не связанную с пунктом меню, можно определить, связав сначала пункт меню с этой клавишей, затем включив этот пункт в свое меню и установив свойство меню *Visible* в *false*. Пункт не будет выводиться, но быстрая клавиша будет работать.

Ваше первое меню

Теперь вы знаете достаточно для того, чтобы создать свое первое меню. Этих знаний, конечно, маловато для того, чтобы сделать пункты меню полезными, скажем, выводить диалоговые окна или работать с буфером обмена, но у нас все впереди. В этой программе обработчики события *Click* просто выводят сообщения с информацией о выбранном пункте меню.

FirsiMainMenu.cs

```
//-------------------------------------------------
// FirstMainMenu.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FirstMainMenu: Form
{
     public static void Main()
     {
          Application.Run(new FirstMainMenu());
     }
     public FirstMainMenu()
     {
          Text = "First Main Menu";

               // Пункты в подменю File

          MenuItem raiOpen = new MenuItem("&Open...",
                                          new EventHandler(MenuFileOpenOnClick),
                                          Shortcut.Ctrl0);

          MenuItem ffliSave = new MenuItem("&Save",
```

см. след. стр.

```
                                new EventHandler(MenuFileSaveOnClick),
                                Shortcut.CtrlS);

        MenuItem miSaveAs = new MenuItem("Save &As...",
                                new EventHandler(MenuFileSaveAsOnClick));

        MenuItem miDash = new MenuItem("-");

        MenuItem ffllExit * new MenuItem("E&xit",
                                new EventHandler(MenuFileExitOnClick));
            // Пункт мен» File

        MenuItem miFile = new MenuItem("&File",
                                new MenuItem[] {miOpen, miSave, miSaveAs,
                                                ffliDasf>, miExit});
            7/ Пункты подменю Edit

        MenuItem miCut = new MenuItem("Cu&t",
                                new EventHandler(MenuEditCutOnClick),
                                Shortcut.CtrlX);

        MenuItem ffliGopy = new MenuItem("&Copy",
                                newEventHandler(MenuEditCopyOnClick),
                                Shortcut.CtrlC);

        MenuItem miPaste * new MenuItem("&Paste",
                                new EventHandler(MenuEditPasteOnClick),
                                Shortcut.CtrlV);
            // Пункт меню Edit

        MenuItem ffliEdit = new MenuItem("&Edit",
                                new MenuItem[] {miCut, miCopy, miPaste});

            // Пункт подмен» Help

        MenuItem miAbout = new MenuItem("&About FirstMainMenu...",
                                newEventHandler(MenuHelpAboutOnClick));
            // Пункт мене Help

        MenuItem miHelp * new MenuItem("&Help",
                                new MenuItem[] {miAbout});
            // Главное меню

        Menu = new MainMenu(new MenuItem[] {miFile, miEdit, miHelp});
    }
    void MenuFileOpenOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("File Open item clicked!", Text);
    }
    void MenuFileSaveOnClick(object obj, EventArgs ea)
```

см. след. стр.

```
    {
        MessageBox.Show("File Save item clicked!", Text);
    }
    void MenuFileSaveAsOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("File Save As item clicked!", Text);
    }
    void MenuFileExitOnClick(object obj, EventArgs ea)
    {
        Close();
    }
    void MenuEditCutOnClick(object obj, EventArgs ea)
    (
        MessageBox.Show("Edit Cut item clicked!", Text);
    }
    void MenuEditCopyOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("Edit Copy item clicked!", Text);
    >
    void MenuEditPasteOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("Edit Paste item clicked!", Text);
    }
    void MenuHelpAboutOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show(Text + " @ 2001 by Charles Petzold");
    }
}
```

Согласен, для программы, просто выводящей несколько сообщений, кода многовато. Но если учесть всю важность меню для большинства приложений, количество кода не кажется несусветным. Большинство действий программы выполняют в ответ на события *Click* меню.

В программе демонстрируются четыре конструктора *MenuItem*. Для пункта меню Open из меню File применяется конструктор с тремя параметрами:

```
MenuItem miOpen = new MenuItem("&Open...",
                       new EventHandler(MenuFileOpenOnClick),
                       Shortcut.CtrlO);
```

Пункт Save As из меню File обычно не имеет быстрой клавиши, поэтому конструктору передаются только два параметра:

```
MenuItem miSaveAs = new MenuItem("Save &As...",
                       new EventHandler(MenuFileSaveAsOnClick));
```

А вот конструктор только с одним строковым параметром:

```
MenuItem miDash = new MenuItem("-");
```

При указании прочерка в качестве пункта подменю, в подменю рисуется горизонтальная линия. Таким образом можно разделять группы пунктов в подменю.

После того, как созданы все пункты подменю File, создается пункт меню верхнего уровня, используя еще одну форму конструктора *MenuItem*:

```
MenuItem miFile = new MenuItem("&File",
                        new MenuItem[] {miQpen, miSave, miSaveAs,
                                        miDash,  miExit  });
```

Конструктору передается текст пункта меню («File») и массив пунктов, появляющихся при вызове подменю.

Далее все то же самое делается для пункта меню Edit (включая подменю, содержащее Cut, Copy и Paste) и Help (содержащего только пункт About). И, наконец, конструктор формы записывает в свойство *Menu* формы объект типа *MainMenu*. В конструкторе *MainMenu* указан массив объектов *MenuItem*, появляющихся на верхнем уровне меню:

```
Menu = new MainMenu(new MenuItem[] {miFile, miEdit, miHelp});
```

В программе FirstMainMenu все обработчики события *Click*, кроме обработчика события для пункта Exit из меню File, выводят сообщения. Обработчик для пункта Exit вызывает метод *Close* класса Form и закрывает форму, прекращая выполнение программы.

Не обязательно устанавливать разные обработчики события для каждого события *Click*. Можно обрабатывать все пункты меню в одном обработчике. Но при использовании разных обработчиков программа становится читабельнее, кроме того, такую программу легче сопровождать. Обрабатывать несколько пунктов меню в одном обработчике события *Click* имеет смысл, только если они составляют некую связанную группу, обычно представляя взаимоисключающие опции.

Необычные меню

Хотя проектирование меню таким образом, чтобы они были похожи на меню других Windows-программ, помогает пользователю ориентироваться в программе, так делать не обязательно. Иногда — скажем, в программе, которую вы пишите сами для себя, — нужен только один пункт меню. Отобразить тот факт, что пункт меню верхнего уровня не вызывает подменю, а выполняет некоторое действие, можно восклицательным знаком.

Эта программа, являющаяся подклассом программы JeuDeTaquin из главы 10, содержит меню из одного пункта «Scramble!»:

JeuDeTaquinWithScramble.cs

```
//--------------------------------------------------------
// JeuDeTaquinWithScramble.cs © 2001 by Charles Petzold
//--------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class JeuDeTaquinWithScramble: JeuDeTaquin
```

см. след. стр.

```
    public new static void Main()
    {
        Application.Run(new JeuDeTaquinWithScramble());
    }
    public JeuDeTaquinWithScramble()
    {
        Menu * new MainMenu(new MenuItem[ ]  {
                    new MenuItem("&Scramble!",
                        new EventHandler(MenuScrambleOnClick)) });
    }
    void MenuScrambleOnClick(object obj, EventArgs ea)
    {
        Randomize();
    }
}
```

Теперь, чтобы перемешать мозаику, не обязательно завершать работу программы.

Одной из первых популярных программ, использующей визуальное иерархическое меню в текстовом режиме MS-DOS, была Lotus 1-2-3- Однако меню Lotus 1-2-3 не содержали всплывающих меню. Меню выводились в одной строке, и каждый следующий уровень меню заменял предыдущий.

В программе Windows Forms можно реализовать такое поведение, определив несколько объектов *MainMenu* и динамически присваивая свойство *Menu* формы равным этим объектам. Этот подход демонстрируется в следующей программе.

OldFashionedMenu.cs

```
//-----------------------------------------------
// QldFashionedItenu.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class OldFashionedMenu: Form
{
    MainMenu mmMain, mmFile, mmEdit;

    public static void Main()
    {
        Application.Run(new OldFashionedMenu());
    }
    public OldFashionedMenu()
    {
        Text = "Old-Fashioned Menu";

        EventHandler eh = new EventHandler(MenuOnClick);

        mmMain = new MainMenu(new MenuItem[]
```

см. след. стр.

```
        {
            new MenuItem("MAIN:"),
            new MenuItem("&File", new EventHandler(MenuFileOnClick)),
            new MenuItem("&Edit", new EventHandler(MenuEditOnClick))
        });

        mmFile = new MainMenu(new MenuItem[]
        {
            new MenuItem("FILE:"),
            new MenuItem("&New", eh),
            new MenuItem("&Open...", eh),
            new MenuItem("&Save", eh),
            new MenuItem("Save &As...", eh),
            new MenuItem("(&Main)", new EventHandler(MenuMainOnClick))
        });

        mmEdit - new MainMenu(new MenuItem[]
        {
            new MenuItem("EDIT:"),
            new MenuItem("Cu&t", eh),
            new MenuItem("&Copy", eh),
            new MenuItem("&Paste", eh),
            new MenuItem("De&lete", eh),
            new MenuItem("(&Main)", new EventHandler(MenuMainOnClick))
        });

        Menu = mmMain;
    }
    void MenuMainOnClick(object obj, EventArgs ea)
    {
        Menu = mmMain; ;
    }
    void MenuFileOnClick(object obj, EventArgs ea)
    {
        Menu - mmFile;
    }
    void MenuEditOnClick(object obj, EventArgs ea)
    {
        Menu = ramEdit;
    }
    void MenuOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("Menu item clicked!", Text);
    }
}
```

Свойства и события *MenuItem*

Единственным (из пяти перечисленных) не рассмотренным ранее конструктором *MenuItem* является конструктор по умолчанию:

```
new MenuItem()
```

Используя его, надо устанавливать свойства объекта *MenuItem*. Свойства, связанные с текстом пункта меню и быстрой клавишей, приведены в таблице:

Свойства *MenuItem* (выборочно)

Тип	Свойство	Доступ
string	*Text*	Чтение/запись
char	*Mnemonic*	Чтение
Shortcut	*Shortcut*	Чтение/запись
bool	*ShowShortcut*	Чтение/запись

Свойство *Mnemonic* указывает символ, следующий за амперсантом в свойстве *Text* (или 0, если мнемонического символа нет). Можно установить свойство *ShowShortcut* в *false* и запретить вывод быстрой клавиши справа от текста пункта меню.

Вот большинство свойств *MenuItem* типа *bool* для чтения и записи:

Свойства *MenuItem* (выборочно)

Тип	Свойство	Доступ
bool	*Visible*	Чтение/запись
bool	*Enabled*	Чтение/запись
bool	*DefaultItem*	Чтение/запись
bool	*Break*	Чтение/запись
bool	*BarBreak*	Чтение/запись

Если установить свойство *Visible* в *false,* пункт не будет появляться в меню. Но быстрая клавиша (если она задана) все равно будет вызывать обработчик события *Click*. Это свойство можно применять для установки быстрых клавиш, не связанных с каким-либо пунктом меню. Кроме того, можно делать какие-то пункты опционально невидимыми для реализации «облегченного» меню, например для новичков,

Возможно, *Enabled* используют чаще прочих свойств из этих пяти. Его устанавливают в *false*, чтобы заблокировать неприменимые в данный момент пункты. Если пункт заблокирован, текст выводится бледным шрифтом, указывая на то, что пункт меню недоступен. Для заблокированного пункта нельзя вызвать обработчик события *Click* ни мышью, ни быстрой клавишей.

Пункты меню File и Edit часто блокируются и разблокируются в зависимости от определенных условий. Обычно пункты Save и Save As заблокированы, когда не загружен ни один документ. Пункты Cut и Copy заблокированы, когда документ загружен, но в нем ничего не выделено. Программа блокирует пункт Paste, если в буфере обмена нет данных, которые могут быть использованы в этой программе. Ниже я объясню, как блокировать пункты меню.

Пункт, свойству *DefaultItem* которого присвоено значение *true*, называется пунктом меню по умолчанию. Текст такого пункта выводится шрифтом полужирного начертания. Для главного меню программы свойство *DefaultItem* имеет смысл устанавливать только для пунктов подменю. При двойном щелчке пункта, активизирующего подменю, вызывается пункт подменю по умолчанию. Например, если в конструктор FirstMainMenu вставить:

```
miExit.DefaultItem = true;
```

двойной щелчок пункта File приведет к закрытию программы. Чаще всего пункты по умолчанию применяются в контекстных меню.

Установка свойства *Break* в *true* приводит к тому, что пункт меню (и все последующие пункты) выводятся в новой колонке. Если свойству *BarBreak* задать *true*, эффект будет точно такой же за исключением того, что между колонками появится вертикальная линия. Хотя обычно так не делают, можно устанавливать свойства *Break* и *BarBreak* для пунктов верхнего уровня главного меню. Эффект от применения этих свойств одинаковый: пункт меню (и все последующие) выводится в новой строке.

Это еще не все свойства *MenuItem*. Вскоре я рассмотрю свойства *Checked* и *RadioCheck*. Кроме того, класс *MenuItem*, как и *MainMenu* и *ContextMenu*, наследует от класса *Menu* важнейший класс *MenuItems* (обратите внимание на множественное число), который я тоже рассмотрю ниже.

В классе *MenuItem* определено пять событий, важнейшее из которых — *Click*. Два из них связаны с прорисовкой владельцем, которую я рассмотрю в конце главы. А вот три других события (включая *Click*):

События *MenuItem* (выборочно)

Событие	Метод	Делегат	Параметры
Click	*OnClick*	*EventHandler*	*EventArgs*
Popup	*OnPopup*	*EventHandler*	*EventArgs*
Select	*OnSelect*	*EventHandler*	*EventArgs*

Обработчики события *Popup* часто устанавливают для пунктов меню верхнего уровня, например File или Edit. Событие *Popup* вызывается перед тем, как выводится подменю, связанное с этим пунктом. Как я уже отмечал, некоторые пункты меню, например Cut, Copy и Paste, блокируются и разблокируются в зависимости от выделенного фрагмента документа или содержимого буфера обмена. Такую блокировку/разблокировку удобно выполнять в обработчике события *Popup*.

Событие *Select* вызывается, когда курсор мыши проходит над пунктом меню или пользователь перемещается между пунктами с помощью клавиш-стрелок. Выбранный пункт обычно выводится другим цветом. Возможно, вы замечали, что некоторые приложения выводят в строке состояния текст, коротко описывающий выделенный пункт меню. Я покажу, как это делается, в главе 20.

Отмеченные пункты меню

Если посмотреть на меню View (Вид) программы Калькулятор (особенно в научном формате), можно увидеть несколько примеров отмеченных пунктов меню. Пункты меню, представляющие выбор одной опции из двух [например, пункт меню Digit Grouping (Группировка цифр)], могут быть отмечены или сброшены наподобие элемента управления *CheckBox*.

Другие группы пунктов меню — вроде Hex (Шестнадцатеричный), Decimal (Десятичный), Octal (Восьмеричный) и Binary (Двоичный) — представляют взаимоисключающие варианты. Выбранный в настоящий момент пункт меню выделен закрашенным кружком — переключателем. (Некоторые Windows-программы

используют галочку для взаимоисключающих пунктов и для пунктов, могущих находиться в состоянии включено/выключено. Использовать галочку для выделения взаимоисключающих пунктов меню последнее время не принято.)

Контролировать появление галочек и переключателей позволяют свойства *MenuItem:*

Свойства *MenuItem* (выборочно)

Тип	Свойство	Доступ
bool	*Checked*	Чтение/запись
bool	*RadioCheck*	Чтение/запись

Задайте *true* свойству *Checked,* чтобы выводилась отметка или переключатель. При этом, если свойству *RadioCheck* задать *true,* будет выводиться переключатель (для обозначения взаимоисключающих вариантов), а если *false* — будет выводиться галочка (для элементов типа включено/выключено).

Следующая программа схожа с программой RadioButtons из главы 12. Главное меню содержит единственный пункт — Format, содержащий 10 элементов: 8 переключателей для выбора цвета, девятый — горизонтальная линяя, а десятый - элемент с текстом Fill, при отметке которого эллипс закрашивается,

CheckAndRadioCheck.cs

```
//-----------------------------------------------
// CheckAndRadioCheck.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CheckAndRadioCheck: Form
{
     MenuItem miColor, miFill;

     public static void Main()
     {
          Application.Run(new CheckAndRadioCheck());
     }
     public CheckAndRadioCheck()
     {
          Text = "Check and Radio Check";
          ResizeRedraw = true;

          string[]    astrColor = {"Black", "Blue",    "Green",  "Cyan",
                                   "Red",   "Magenta", "Yellow", "White"};
          MenuItem[]  ami       = new MenuItem[astrColor.Length + 2];
          EventHandler ehColor  = new EventHandler(MenuFormatColorOnClick);

          for (int i = 0; i < astrColor.Length; i++)
          {
```

см. след. стр.

```
            ami[i] - new MenuItem(astrColor[i], ehColor);
            ami[i].RadioCheck = true;
        }
    : fliiColor = ami[0];
        miColor.Checked = true;

        ami[astrColor.Length] = new MenuItem("-");

        miFill = new MenuItem("&Fill",
                              new EventHandler(MenuFormatFillOnClick));

        ami[astrColor.Length * 1] = miFill;

        MenuItem mi = new MenuItem("&Format", ami);

        Menu = new MainMenu(new MenuItem[] {mi});
    }
    void MenuFormatColorOnClick(object obj, EventArgs ea)
    {
        miColor.Checked = false;
        miColor = (MenuItem)obj;
        miColor.Checked - true;

        Invalidate();
    }
    void MenuFormatFillOnClick(object obj, EventArgs ea)
    {
        MenuItem ral = (MenuItem)obj;

        mi.Checked ^= true;

        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        ••Graphics grfx* pea.Graphics;

        if (miFill.Checked)
        {
            Brush brush = new SolidBrush(Color.FromName(miColor.Text));
            grfx.FillEllipse(brush, 0, 0, ClientSize.Width - 1,
                                          ClientSize.Height - 1);
        }
        else
        {
            Pen pen = new Pen(Color.FromName(miColor.Text));
            grfx.DrawEllipse(pen, 0, 0, ClientSize.Width - 1,
                                        ClientSize.Height - 1);
        }
    }
}
```

В конструкторе сначала определяется массив из 8 текстовых строк, соответствующих названиям цветов, а затем массив структур *MenuItem* достаточного размера для хранения этих 8 названий цветов и еще двух пунктов меню:

```
MenuItem[] ami = new MenuItem[astrColor.Length + 2];
```

Пункты из группы взаимоисключающих обычно связаны с одним обработчиком события *Click*. Поэтому обработчик события определяется до создания пунктов меню:

```
EventHandler ehColor = new EventHandler(MenuFormatColorOnClick);
```

В цикле *for* создаются 8 пунктов меню на основе 8 названий цветов и обработчика событий *ehColor*. Свойство *RadioCheck* устанавливается в *true,* так что ксгда свойство *Checked* равно *true,* вместо галочки выводится переключатель.

Переменная *miColor,* хранящаяся в поле, указывает, какой объект *MenuItem* отмечен в настоящее время. Конструктор задает это поле равным первому объекту *MenuItem* в массиве и устанавливает свойство *Checked* в *true.*

```
miColor = ami[0];
miColor.Checked = true;
```

Рассмотрим обработчик *MenuFormatColorOnClick* и затем вернемся к конструктору. В начале обработчик *Click* снимает отметку с отмеченного в настоящий момент пункта меню;

```
miColor.Checked = false;
```

Затем поле *miColor* приравнивается первому аргументу обработчика событий, представляющему собой щелкнутый пункт меню;

```
miColor = (MenuItem)obj;
```

Затем обработчик события отмечает пункт меню;

```
miColor.Checked = true;
```

и делает форму недействительной, что приводит к перерисовке эллипса. В этом блоке кода показан обычный способ отмечать и снимать отметки со взаимоисключающих пунктов меню.

Вернемся к конструктору. После создания 8 пунктов, соответствующих 8 цветам, создается девятый — горизонтальная разделительная линия, а затем десятый пункт — Fill:

```
miFill = new MenuItem("&Fill",
                      new EventHandler(MenuFormatFillOnClick));

ami[astrColor.Length + 1] = miFill;
```

Переменная *miFill* тоже хранится в поле. Метод *OnPaint* при рисовании эллипса обращается как к *miColor,* так и к *miFill.*

Однако методу *MenuFormatFillOnClick* не требуется доступ к *miFill* Обработчик события получает объект *MenuItem,* который щелкнул пользователь, приводя первый аргумент:

```
MenuItem mi = (MenuItem)obj;
```

а затем переключает состояние этого пункта:

```
mi.Checked "= true;
```

Два предыдущих выражения можно заменить одним оператором:

```
miFill.Checked "= true;
```

но из соображений написания более «общих» обработчиков не следует обращаться к конкретному пункту меню в обработчике события. Даже если вы добавите другие пункты меню типа включено/выключено, влияющие на обработку *OnPaint*, все равно можно будет использовать тот же общий подход к установке/снятию отметок.

Метод *OnPaint* в программе CheckAndRadioCheck преобразует название цвета из пункта меню в объект *Color* с помощью статического метода *Color.FromName* во время подготовки к созданию кисти или пера:

```
Color.FromName(miColor.Text)
```

Ясно, что не каждый пункт меню содержит свойство *Text*, которое может быть преобразовано в нужный объект. Подобное использование текста пункта меню — вообще не очень хорошая идея. Во-первых, при переводе меню на другой язык, новый текст вряд ли так легко преобразуется в объект *Color*. Во-вторых, в названия цветов неудобно вставлять амперсанты, так как их приходится удалять перед передачей текста методу *Color.FromName*.

Чуть позже я продемонстрирую более общий подход к обработке взаимоисключающих пунктов меню. (Вы спросите, заключается ли он в применении классов-наследников *MenuItem*? А вы как думаете?)

Работа с контекстными меню

Контекстные меню в некотором смысле проще, чем главные, в основном оттого, что они меньше и иногда содержат только список пунктов меню без подменю. В этой программе создается контекстное меню, позволяющее пользователю выбрать цвет фона формы:

ContextMenuDemo.cs

```
//-----------------------------------------------
// ContextMenuDemo.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ContextMenuDemo : Form
{
     MenuItem miColor;

     public static void Main()
     {
```

см. след. стр.

```
        Application.Run(new ContextMenuDemo());
    }
    public ContextHenuDemo()
    {
        Text = "Context Menu Demo";

        EventHandler eh = new EventHandler(MenuColorOnClick);

        MenuItem[] ami = { new MenuItem("Black",   eh),
                           new MenuItem("Blue",    eh),
                           new MenuItem("Green",   eh),
                       •;• new MenuItem("Cyan",    eh),
                           new MenuItem("Red",     eh),
                           new MenuItem("Magenta", eh),
                           new MenuItem("Yellow",  eh),
                           new MenuItem("White",   eh) };

        foreach (MenuItem mi in ami)
            mi.RadioCheck = true;

        raiColor = ami[3];
        miColor.Checked - true;
        BackColor = Color.FromName(miColor.Text);

        ContextMenu = new ContextMenu(ami);
    }
    void MenuColorOnClick(object obj, EventArgs ea)
    {
        miColor.Checked = false;
        miColor = (MenuItem) obj;
        miColor.Checked = true;

        BackColor = Color.FromName(miColor.Text);
    }
}
```

В этой программе 8 пунктов меню определяются прямо при инициализации массива *ami* типа *MenuItem*. Как и в предыдущей программе, все пункты связаны с одним обработчиком события *Click*. В цикле/or свойство *RadioCheck* для каждого пункта устанавливается в *true*. Затем конструктор присваивает переменную-поле *miColor* равной четвертому элементу массива, задает отметку для этого пункта и в соответствии с отмеченным пунктом устанавливает цвет фона.

Конструктор завершается созданием нового объекта *ContextMenu* из массива объектов *MenuItem* и присваиванием свойства формы *ContextMenu* равным этому объекту:

```
ContextMenu = new ContextMenu(ami);
```

Контекстное меню можно вызвать, щелкнув правой кнопкой в любом месте клиентской области. Кроме того, класс *ContextMenu* содержит метод, позволяющий вывести контекстное меню, не задавая свойство элемента управления *ContextMenu:*

Методы *ContextMenu* (выборочно)

void Show(Control ctrl, Point ptLocation)

Этот метод можно использовать, если нужно выводить разные контекстные меню в зависимости от того, в каком месте элемента управления (или формы) пользователь щелкнул мышью.

С точки зрения преобразования текста пункта меню в цвет, программа ContextMenuDemo не отличается от CheckAndRadioCheck. При реализации взаимоисключающих пунктов меню, связанных с одним и тем же обработчиком, гораздо правильнее использовать в меню свой класс-наследник *MenuItem*. Он включает новое поле или свойство, в котором хранится связанный с этим пунктом объект.

В следующей программе класс *MenuItemColor* является наследником *MenuItem*. Класс содержит закрытое поле *clr*, в котором хранится объект *Color,* Открытое свойство *Color* позволяет другим классам обращаться к этому цвету. Кроме того, новый класс содержит новый конструктор, позволяющий создать объект *MenuItemColor* с определенным цветом. Эта программа похожа на ContextMenuDemo за исключением того, что в ней вместо *MenuItem* используется *MenuItemColor,*

BetterContextMenu.cs

```
//---------------------------------------------
// BetterContextMenu.cs © 2001 by Charles Petzold
//         ---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BetterContextMenu: Form
{
     MenuItemColor micColor;

     public static void Main() '• •
     {
          Application.Run(new BetterContextMenu());
     }
     public BetterContextMenu()
     {
          Text в "Better Context Menu Demo";

          EventHandler eh * new EventHandler(MenuColorOnClick);

          MenuItemColor[] aaic =
          {
               new MenuItemColor(Color.Black,   "&Black",   eh),
               new MenuItemColor(Color.Blue,    "B&lue",    eh),
               new MenuItemColor(Color.Green,   "&Green",   eh),
               new MenuItemColor(Color.Cyan,    "&Cyan",    eh),
```

см. след. стр.

```
            new MenuItemColor(Color.Red,     "&Red",     eh),
            new MenuItemColor(Color.Magenta, "&Magenta", eh),
            new MenuItemColor(Color.Yellow,  "&Yellow",  eh),
            new MenuItemColor(Color.White,   "&White",   eh)
        };

        foreach (MenuItemColor rale in amic)
            mic.RadioCheck = true;

        micColor = amic[3];
        micColor.Checked - true;
        BackColor = fliicColor. Color;

        ContextMenu = new ContextMenu(amic);
    }
    void MenuColorOnClick(object obj, EventArgs ea)
    {
        micColor.Checked = false;
        micColor = (MenuItemColor) obj;
        micColor.Checked * true;

        BackColor = micColor.Color;
    }
}
class MenuItemColor: MenuItem
{
    Color clr;

    public MenuItemColor(Color clr, string str, EventHandler eh):
                                                        base(str, eh)
    {
        Color = clr;
    }
    public Color Color
    {
        get { return clr; }
        set { clr = value; }
    }
}
```

Теперь свойство *BackColor* формы можно приравнять свойству *Color* отмеченного в настоящее время объекта *MenuItemColor*.

Набор пунктов меню

В программе FirstMainMenu меню строилось снизу вверх. Сначала создавались объекты *MenuItem*для самых последних пунктов (типа Open, Save и т. д.). Они затем объединялись в массивы для создания пунктов верхнего уровня (File, Edit и т. д.), А пункты верхнего уровня собирались вместе в объект *MainMenu*.

Программа была бы стройнее и легче в сопровождении, если бы меню создавалось сверху вниз, начиная с объекта *MainMenu*, затем добавлялись бы объекты *MenuItem*, соответствующие пунктам меню верхнего уровня (File, Edit и т. д.), и, наконец, добавлялись бы пункты в подменю (Open, Save и т. д.).

Пара важных свойств, определенных в классе *Menu,* делает такой подход возможным. Как я уже говорил, классы *MenuItem*, *MamMenu* и *ContextMenu* порождены *Menu* и наследуют эти свойства:

Свойства *Menu* (выборочно)

Тип	Свойство	Доступ
boo!	*IsParent*	Чтение
MenuMenuItemCollection	*MenuItems*	Чтение

Вам ничего не напоминает *MenuMenuItemCollection?* Этот класс очень похож на *ImageList.ImageCollection* из главы 11 и *Control.ControlCollection* из главы 12. Как и в них, в *MenuMenuItemCollection* реализованы интерфейсы *IList*, *Icollection* и *IEnumerable*. К свойству *MenuItems* можно обращаться по индексу, как если бы это был массив объектов *MenuItem*. Можно вызывать методы *Add*, *Remove* и *Clear*.

Свойство *IsParent* указывает, является ли пункт меню родительским по отношению к другим пунктам и, следовательно, допустимо ли свойство *MenuItems*.

Кроме того, класс *MenuItem* содержит такие связанные свойства:

Свойства *MenuItem* (выборочно)

Тип	Свойство	Доступ
Menu	*Parent*	Чтение
int	*Index*	Чтение/запись

Свойство *Parent* указывает на родительское меню конкретного пункта меню, свойство *Index* (доступное для записи) указывает на индекс (начиная с 0) объекта *MenuItem* внутри конкретного подменю.

В классе *MenuMenuItemCollection* реализованы методы, позволяющие добавлять дочерние пункты меню к главному или контекстному меню либо другому пункту:

Методы *Menu.MenuItemCollection* (выборочно)

```
MenuItem Add(string strText)
MenuItem Add(string strText, EventHandler ehClick)
MenuItem Add(string strText, MenuItem[] ami)
int Add(MenuItem mi)
int Add(int index, MenuItem mi)
void AddRange(MenuItem[] ami)
```

Для определения количества пунктов меню в коллекции и доступа к ним можно использовать следующие свойства (второе из которых индексатор):

Свойства *Menu.MenuItemCoHection*

Тип	Свойство	Доступ
int	*Count*	Чтение
MenuItem	*[J*	Чтение

Пункты меню можно искать и удалять:

Методы *Menu.MenuItemCoHection* (выборочно)

```
bool Contains(MenuItem mi)
int  IndexOf(MenuItem  mi)
void  Remove(MenuItem  mi)
void  RemoveAt(int  index)
void  Clear()
```

В следующей версии программы ContextMenuDemo — ContextMenuAdd — для создания меню применяются метод *Add* и возможность индексирования в свойстве *MenuItem*. Вначале создается объект *ContextMenu*, и к нему добавляются пункты меню.

ContextMenuAdd.cs

```
//-----------------------------------------
// ContextMenuAdd.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ContextMenuAdd: Form
{
     MenuItem miColor;

     public static void Main()
     {
          Application.Run(new ContextMenuAdd());
     }
     public ContextMenuAdd()
     {
          Text = "Context Menu Using Add";

          ContextMenu  cm = new ContextMenu();
          EventHandler eh = new EventHandler(MenuColorOnClick);

          cm.MenuItems.Add("Black",   eh);
          cm.MenuItems.Add("Blue",    eh);
          cm.MenuItems.Add("Green",   eh);
          cm.MenuItems.Add("Cyan",    eh);
          cm.MenuItems.Add("Red",     eh);
          cm.MenuItems.Add("Magenta", eh);
```

см, след. стр.

```
        т. MenuItems.Add("Yellow",    eh);
        cm.MenuItems.Add("White",     eft);

        foreach (MenuItem mi in cm.MenuItems)
             mi.RadioCheck = true;

        miColor = cm.MenuItems[3];
        miColor.Checked = true;
        BackColor - Color.FromName(miColor.Text);

        ContextMenu = cm;
    }
    void MenuColorOnClick(object obj, EventArgs ea)
    {
        miColor.Checked =false;
        miColor = (MenuItem) obj;
        miColor.Checked = true;

        BackColor = Color.FromName(miColor.Text);
    }
```

Можно было бы уменьшить размер исходного кода программы, приравняв свойство *ContextMenu* формы новому объекту *ContextMenu*:

```
ContextMenu = new ContextMenu();
```

Для добавление пунктов тогда можно было бы использовать примерно такие выражения:

```
ContextMenu.MenuItems.Add("Black", eh);
```

Ранее я преобразовал ContextMenuDemo в программу BetterContextMenu, определив класс *MenuItemColor*, являющийся наследником *MenuItem*, но хранящий объект *Color*. Как же преобразовать ContextMenuAdd, чтобы в ней задействовать объекты *MenuItemColor*?

Очевидно, что нельзя применять такой же метод *Add*, что и в ContextMenuAdd. Вызов:

```
cm. MenuItems.Add("Black", eh);
```

неявно создает новый объект типа *MenuItem* и добавляет его в набор пунктов меню. Результат следующего выражений тот же, но объект создается в явном виде:

```
cm.MenuItems. Add(new MenuItem("Black", eh));
```

Чтобы преобразовать программу с использованием класса *MenuItemColor*, нужно выполнять примерно такие вызовы;

```
cm.MenuItems. Add(new MenuItemColor(Color. Black, "Black", eh»;
```

Следующая программа содержит единственный пункт меню верхнего уровня Facename и добавляет все доступные названия шрифтов в это меню во время обработки события *Popup*. Теоретически это лучше, чем создавать меню при запус-

ке программы, так как тогда появляется возможность выводить имена шрифтов, добавленных после запуска программы.

FontMenu.cs

```
//-------------------------------------------
// FontMenu.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FontMenu: Form
{
     const int iPointSize = 24;
     string    strFacename;

     public static void Main()
     {
          Application.Run(new FontMenu());
     }
     public FontMenu()
     {
          Text = "Font Menu";

          strFacename = Font.Name;

          Menu = new MainMenu();

          MenuItem mi = new MenuItem("&Facename");
          mi.Popup += new EventHandler(MenuFacenameOnPopup);
          mi.MenuItems.Add(" ");
          Menu.MenuItems.Add(mi);
     }
     void MenuFacenameOnPopup(object obj, EventArgs ea)
     {
          MenuItem     miFacename = (MenuItem)obj;
          FontFamily[] aff        * FontFamily.Families;
          EventHandler ehClick    = new EventHandler(MenuFacenameOnClick);
          MenuItem[]   ami        * new MenuItem[aff.Length];

          for (int i = 0; i < aff.Length; i++)
          {
               ami[i] * new MenuItem(aff[i].Name);
               ami[i].Click += ehClick;

               if (aff[i].Name == strFacename)
                    ami[i].Checked = true;
          }
          miFacename.MenuItems.Clear();
```

см. след. стр.

```
            miFacename.MenuItems.AddRange(ami);
        }
        void MenuFacenameOnClick(object obj, EventArgs ea)
        {
            MenuItem mi = (MenuItem)obj;
            strFacename = mi.Text;
            Invalidate();
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics grfx * pea.Graphics;
            Font     font - new Font(strFacename, iPointSize);

            StringFormat strfmt  = new StringFormat();
            strfmt.Alignment     = StringAlignment.Center;
            strfmt.LineAlignment = StringAlignment.Center;

            grfx.DrawString("Sample Text", font, new SolidBrush(ForeColor),
                            ClientRectangle, strfmt);
        }
    }
```

При определении меню в конструкторе в подменю добавляется один пустой пункт. Для вызова события *Popup* R подменю требуется наличие хотя бы одного пункта,

Обработчик события *MenuFacenameOnPopup* начинается с получения пункта меню верхнего уровня:

```
MenuItem miFacename = (MenuItem)obj;
```

Затем метод получает массив доступных названий шрифтов с помощью статического метода *FontFamily.Families* Затем определяется массив *MenuItem* такого размера и инициализирует его элементы. попутно устанавливая свойство *Check* пункта, соответствующего названию шрифта из поля *strFacename.*

После инициализации массива *MenuItem* метод завершается очисткой всех элементов всплывающего меню и добавлением нового массива пунктов:

```
miFacename.MenuItems.Clear();
miFacename.MenuItems.AddRange(ami);
```

С целью уменьшения размера исходного кода в программе не обрабатывается ситуация, возникающая, если будет выбран шрифт, к которому неприменим стиль *FontStyle.Regular* (например, Aharoni). Если выбрать такой шрифт, конструктор *Font* в методе *OnPaint* сгенерирует исключение, и программа завершится. (Подробнее об этом и о работе со шрифтами см. главу 9.)

Но настоящая проблема в этой программе — размер подменю. Вполне вероятно, что высота списка будет больше высоты дисплея. Очевидно, для выбора шрифта больше подходят диалоги, и я это покажу в главе 16.

Стандартные меню (предложение)

Чтобы облегчить чтение и сопровождение программы, следует создавать меню в том же порядке, в котором они видны пользователю. Таким образом, конструктор вначале должен создавать первый пункт меню верхнего уровня (обычно File), затем все пункты подменю File (обычно New, Open, Save и т. д.), затем следующий пункт меню верхнего уровня (обычно Edit), затем пункты в подменю Edit и в самом конце — пункт About подменю Help.

Экспериментируя с созданием пунктов меню верхнего уровня с помощью метода *Add* класса *MenuMenuItemCollection*, вы заметите, что разные версии метода *Add* неодинаковы. Некоторые полезней других. Вы можете выработать собственный стиль создания главного меню, используя только подмножество конструкторов *MenuItem* и методов *Add*. Я предлагаю задействовать только одну форму конструктора *MenuItem* и одну форму метода *Add*. Вы вполне можете пренебречь этим предложением. Я и сам пренебрегаю им в дальнейших главах. Но мне бы хотелось исследовать некоторые моменты, связанные с созданием меню.

Вначале рассмотрим создание некоего меню верхнего уровня без обработчиков и быстрых клавиш. Код выглядит примерно так:

```
Menu = new MainMenu();
Menu.MenuItems.Add("&File");
Menu.MenuItems[0].MenuItems.Add("&Open...");
Menu.MenuItems[0].MenuItems.Add("&Save...");
⋮
Menu.MenuItems.Add("&Edit");
Menu.MenuItems[1].MenuItems.Add("Cu&t");
Menu.MenuItems[1].MenuItems.Add("&Copy");
⋮
```

Этот код хорошо структурирован (если не сказать, что он замечательный). Пункты меню верхнего уровня (File и Edit) добавляются к объекту *MainMenu* методом *Add* свойства *MenuItems*. Выражение *MenuMenuItems[0]* относится к пункту File, а *MenuMenuItems[1]* — к пункту Edit. Каждый из этих пунктов имеет собственное свойство *MenuItems*, представляющее набор пунктов подменю. Пункты типа Open, Save, Cut, Copy и т. д. добавляются методом *Add* этого свойства.

Кроме взаимоисключающих, большинство пунктов меню связано с собственными обработчиками события *Click*. Чтобы меню могло делать что-то полезное, придется преобразовать предыдущие выражения в примерно такие:

```
Menu.MenuItems[0].MenuItems.Add("&Open...",
                        new EventHandler(MenuFileOpenOnClick));
```

Но пункт меню Open обычно включает быструю клавишу Ctrl+O, а метода *Add*, принимающего в качестве аргумента быструю клавишу, нет. Вам придется вставить дополнительно выражение вроде:

```
Menu.MenuItems[0].MenuItems[0].Shortcut = Shortcut.CtrlO;
```

для задания свойства *Shortcut* пункта меню Open. При этом, если позже вы измените свой код и вставите пункт New перед Open, выражение придется изменить, так как индекс изменится:

```
Menu.MenuItems[0].MenuItems[1].Shortcut = Shortcut.Ctrl0;
```

Тупик! Я *думаю,* вы согласитесь с тем, что установка свойства пункта меню не должна требовать указания двух уровней свойств *MenuItems.*

Возможно, гораздо лучше было бы вначале определить *MenuItem* примерно так

```
miFileOpen = new MenuItem("&Open",
                  new EventHandler(MenuFileOpenOnClick),
                  Shortcut.Ctrl0);
```

а затем добавить этот пункт меню в набор *MenuItems:*

```
Menu.MenuItems[0].MenuItems.Add(miFileOpen);
```

Так как создание *miFileOpen* разбивается на несколько строк кода, возможно, проще будет использовать простой конструктор и явно присвоить свойства *MenuItem:*

```
miFileOpen = new MenuItem("&Open");
miFileOpen.Click += new EventHandler(MenuFileOpenOnClick);
miFileOpen.Shortcut = Shortcut.Ctrl0;
Menu.MenuItems[0].MenuItems.Add(miFileOpen);
```

А есть ли вообще смысл использовать версию *Add,* принимающую единственный аргумент типа *string?* Она может быть полезна при добавлении горизонтальной разделительной строки в меню:

```
Menu.MenuItems[0].MenuItems.Add("-");
```

А еще — при добавлении пунктов меню верхнего уровня:

```
Menu.MenuItems.Add("F&ormat");
```

При этом многие пункты меню верхнего уровня требуют наличия обработчиков события *Popup,* в которых можно было бы блокировать/разблокировать пункты подменю. И снова имеет смысл вначале создать объект *MenuItem,* а затем добавить созданный пункт к меню:

```
MenuItem mi = new MenuItem("&File");
mi.Popup += new EventHandler(MenuFileOnPopup);
Menu.MenuItems.Add(mi);
```

Чтобы быть абсолютно последовательным при создании меню, понадобится всего одна форма конструктора *MenuItem* (версия с одним параметром типа *siring)* и одна версия метода *Add* (с параметром типа *MenuItem).*

Как вы могли заметить, я предпочитаю сводить количество переменных-полей класса к минимуму. Menu Designer из Visual Studio .NET делает каждый объект *MenuItem* полем. Это абсолютно не нужно! Так как большинство пунктов меню имеет собственные обработчики события *Click,* вовсе не обязательно сохранять все объекты *MenuItem.* Исключение составляют пункты, которые должны быть блокированы/разблокированы при обработке события *Popup.* Эти пункты могут храниться в полях.

Наконец, я хочу обратить ваше внимание на маленькую переменную типа *int,* которую я называю *index.* Каждый раз, когда вы добавляете пункт меню верхнего уровня, используя выражение вроде:

```
Menu.MenuItems.Add(mi);
```

вы можете вычислить новое значение переменной *index*:

```
index = Menu.MenuItems.Count - 1;
```

Эту переменную можно использовать для добавления пунктов к подменю, например так:

```
Menu.MenuItems[index].MenuItems.Add(miFileOpen);
```

Для меню File использовать эту переменную вместо явного 0 едва ли необходимо: File будет первым пунктом главного меню отныне и вовеки веков. Но для других пунктов эта переменная может быть весьма полезна, особенно если однажды вы захотите изменить конструктор и вставить новое подменю.

Следующая программа демонстрирует мой подход к созданию стандартных меню последовательно сверху вниз с помощью единственной формы конструктора *MenuItem* и единственной формы метода *Add*.

StandardMenu.cs

```
//---------------------------------------
// StandardMenu.cs © 2001 by Charles Petzold
//. . . . . .-----------. . . . . —
using System;
using System.Drawing;
using System.Windows.Forms;

class StandardMenu: For»
{
     MenuItem ffliFileOpen, miFileSave;
     MenuItem miEditCut, miEditCopy, miEditPaste;

          // Экспериментальные переменные для кода Popup.

     bool bDocumentPresent  = true;
     bool bNonNullSelection = true;
     bool bStuffInClipboard - false;

     public static void Main()
     {
          Application.Run(new StandardMenu());
     }
     public StandardMenu()
     {
          Text = "Standard Menu";
          Menu = new MainMenu();

               // File(Файл)

          MenuItem mi = new MenuItem("&File");
          mi.Popup += new EventHandler(MenuFileOnPopup);
```

см. след. стр.

```
Menu.MenuItems.Add(mi);
int index = Menu.MenuItems.Count - 1;

    // File Open(Открыть)

miFileOpen = new MenuItem("&Open...");
miFileOpen.Click += new EventHandler(MenuFileOpenOnClick);
fltiFlleOpen.Shortcut = Shortcut.Ctrl0;
Menu.MenuItems[index].MenuItems.Add(miFileOpen);

    // File Save(Сохранить)

miFileSave  = new MenuItem("&Save");
miFileSave.Click += new EventHandler(MenuFileSaveOnClick);
miFileSave.Shortcut = Shortcut.CtrlS;
Menu.MenuItems[index].MenuItems.Add(miFileSave);

    // Горизонтальная линия.

mi = new MenuItem("-");
Menu.MenuItems[index].MenuItems.Add(mi);

    // File Exit(Выход)

mi = new MenuItem("E&xit");
mi.Click += new EventHandler(MenuFileExitOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

    // Edit(Правка)

mi = new MenuItem("&Edit");
mi.Popup += new EventHandler(MenuEditOnPopup);
Menu.MenuItems.Add(mi);
index = Menu.MenuItems.Count - 1;

    // Edit Cut(Вырезать)

miEditCut * new MenuItem("Cu&t");
ffliEditCut.Click += new EventHandler(MenuEditCutOnClick);
miEditCut.Shortcut - Shortcut.CtrlX;
Menu.MenuItems[index].MenuItems.Add(miEditCut);

    // Edit Copy(Копировать)

raiEditCopy = newMenuItem("&Copy");
miEditCopy.Click += new EventHandler(MenuEditCopyOnClick);
miEditCopy.Shortcut = Shortcut.CtrlC;
Menu.MenuItems[index].MenuItems.Add(miEditCopy);

    // Edit Paste(Вставить)
```

См. след. стр.

```
        miEditPaste = new MenuItem("&Paste");
        miEditPaste.Click += new EventHandler(MenuEditCopyOnClick);
        miEditPaste.Shortcut = Shortcut.CtrlV;
        Menu.MenuItems[index].MenuItems.Add(miEditPaste);

            // Help(Справка)

        mi = new MenuItem("&Help");
        Menu.MenuItems.Add(mi);
        index = Menu.MenuItems.Count - 1;

            // Help About(О программе)

        ffii = new MenuItem("&About StandardMenu...");
        mi.Click += new EventHandler(MenuHelpAboutOnClick);
        Menu.MenuItems[index].MenuItems.Add(mi);
    }
    void MenuFileOnPopup(object obj, EventArgs ea)
    {
        ffiiFileSave.Enabled = bDocumentPresent;
    }
    void MenuEditOnPopup(object obj, EventArgs ea)
    {
        miEditCut.Enabled = bNonNullSelection;
        miEditCopy.Enabled = bNonNullSelection;
        miEditPaste.Enabled = bStuffInClipboard;
    }
    void MenuFileOpenOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("This should be a File Open dialog box!", Text);
    }
    void MenuFileSaveOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("This should be a File Save dialog box!", Text);
    }
    void MenuFileExitOnClick(object obj, EventArgs ea)
    {
        Close();
    }
    void MenuEditCutOnClick(object obj, EventArgs ea)
    {
        // Копировать выделенный фрагмент в буфер обмена, удалить из документа.
    }
    void MenuEditCopyOnClick(object obj, EventArgs ea)
    {
        // Копировать выделенный фрагмент в буфер обмена.
    }
    void MenuEditPasteOnClick(object obj, EventArgs ea)
    {
        // Копировать данные из буфера обмена в документ.
```

см. след. стр.

```
    }
    void MenuHelpAboutOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("StandardMenu © 2001 by Charles Petzold", Text);
    }
}
```

Хотя это и не самый короткий код из всех возможных, мне кажется, что в этой программе достигнут определенный уровень ясности и ее легко будет сопровождать.

Прорисовка владельцем

Мы уже почти достигли конца главы, а я все еще не объяснил, как вставлять в пункты меню маленькие картинки или как выводить текст пунктов разными шрифтами и цветами.

Все, что можно сделать с меню, помимо уже рассмотренных методик, требует применения *прорисовки владельцем*. Для каждого объекта *MenuItem*, который вы хотите рисовать самостоятельно, надо установить следующее свойство в *true*:

Свойства *MenuItem* (выборочно)

Тип	Свойство	Доступ
boot	*OwnerDraw*	Чтение/запись

Обычно его задают только для пунктов всплывающих меню. Если определить *OumerDraw* как *true,* нужно установить обработчик двух событий:

События *MenuItem* (выборочно)

Событие	Метод	Делегат	Аргумент
MeasureItem	*OnMeasureItem*	*MeasureItemEventHandler*	*MeasureItemEventArgs*
DrawItem	*OnDrawItem*	*DrawItemEventHandler*	*DrawItemEventArgs*

При подготовке к рисованию пункта меню (обычно перед выводом всплывающего меню) Windows вызывает обработчик события *MeasureItem*. Событию соответствует объект типа *MeasureItemEventArgs*.

Свойства *MeasureItemEventArgs*

Тип	Свойство	Доступ
int	*Index*	Чтение
Graphics	*Graphics*	Чтение
int	*ItemWidth*	Чтение/запись
int	*ItemHeight*	Чтение/запись

Обработчик события *MeasureItem* получает свойства *ItemWidth* и *ItemHeight*, равные 0. Вы должны приравнять их ширине и высоте рисуемого пункта меню. Свойство *Index* позволяет определить, для какого пункта меню вычисляются размеры. При необходимости свойство *Graphics* получит разрешение устройства в точках на дюйм или размер текстовых элементов с помощью *MeasureString*.

Вскоре Windows вызывает обработчики события *DrawItem*, которому соответствует объект *DrawItemEventArgs*:

Свойства *DrawItemEventArgs*

Тип	Свойство	Доступ
int	*Index*	Чтение
Graphics	*Graphics*	Чтение
Rectangle	*Bounds*	Чтение
DrawItemState	*State*	Чтение
Font	*Font*	Чтение
Color	*BackColor*	Чтение
Color	*ForeColor*	Чтение

Задача программы — нарисовать пункт меню, используя объект *Graphics* внутри прямоугольника, определенного свойством *Bounds*. Не следует полагать, что левый верхний угол свойства *Bounds* расположен в точке с координатами (0, 0)! На самом деле прямоугольник *Bounds* находится внутри всплывающего меню.

Ширина прямоугольника больше, чем величина, указанная во время обработки события *MeasureItem*, чтобы вместить отметку стандартного размера слева от пункта меню.

Перечисление *DrawItemState* указывает, является ли пункт меню выделенным, заблокированным или отмеченным:

Перечисление *DrawItemState*

Член	Значение
None	0
Selected	1
Grayed	2
Disabled	4
Checked	8
Focus	16
Default	32
HotLight	64
Inactive	128
NoAccelerator	256
NoFocusRect	512
ComboBoxEdit	4096

Некоторые из членов этого перечисления применимы к другим элементам управления, для которых существует возможность прорисовки владельцем.

Обычно свойство *BackColor* объекта *DrawItemEventArgs* равно *SystemColors.Window*, а свойство *ForeColor* — *SystemColors.WindowText*. Чтобы меню было совместимо с обычными меню, не используйте эти цвета! Вместо этого применяйте *SystemColors.Menu* и *SystemColors.MenuText*. Если пункт выбран, *BackColor* равно *SystemColors.Highlight*, а *ForeColor* — *SystemColors.HighlightText*. Это правильные цвета для выбранных пунктов.

Свойство *Font* свойства *DrawItemEventArgs* равно *SystemInformation.MenuFont*. *DrawItemEventArgs* содержит также два метода, помогающие нарисовать элемент:

Методы *DrawItemEventArgs*

```
void DrawBackground();
void DrawFocusRectangle();
```

Метод *DrawFocusRectangle* не используется для пунктов меню,

Для рисования стрелок, переключателей и галочек полезным будет статический метод класса *ControlPaint*.

Статические методы *DrawMenuGlyph* класса *ControlPaint*

```
void DrawMenuGlyph(Graphics grfx, Rectangle rect,
                   HenuGlyph mg)
void DrawMenuGlyph(Graphics grfx, int x, int y, int cx, int cy,
                   MenuGlyph mg)
```

MenuGlyph представляет собой еще одно перечисление:

Перечисление *MenuGlyph*

Член	Значение
Min	0
Arrow	0
Checkmark	i
Bullet	2
Max	2

Измерять размер пункта меню можно по-разному. Нормальный шрифт, применяемый для вывода текста пунктов (как я упоминал), доступен из *SystemInformationMenuFont*. Другая важная величина хранится в *SystemInformation.MenuCheckSize* и представляет собой ширину и высоту галочки по умолчанию. Как видите, для статического метода *ControlPaint.DrawMenuGlyph* нужно указать высоту и ширину глифа (например, галочки). Если ваши пункты меню выше, чем нормальные, и вы хотите применять галочки, возможно, вам потребуется масштабировать их. Это приведет к тому, что придется учитывать ширину масштабированной галочки при измерении размера пункта в процессе обработки события *MeasureItem*.

Следующая программа содержит единственный пункт верхнего уровня — Facename. Во всплывающем меню три пункта, содержащих три самых распространенных шрифта. С помощью прорисовки владельцем эти пункты меню выводятся шрифтами, соответствующими их названиям,

OwnerDrawMenu.cs

```
//-----------------------------------------
// OwnerDrawMenu.cs © 2001 by Charles Petzold
//-----------------------------------------
```

см. след. стр.

```
using System;
using System.Drawing;
using System.Drawing.Text;     // Для перечисления HotkeyPrefix
using System.Windows.Forms;

class OwnerDrawMenu: Form
{
     const    int  iFontPointSize * 18; // Для пунктов меню
     MenuItem     miFacename;

     public static void Main()
     {
          Application.Run(new OwnerDrawMenu());
     }
     public OwnerDrawMenu()
     {
          Text = "Owner-Draw Menu";

               // Пункты верхнего уровни

          Menu = new MainMenu();
          Menu.MenuItems.Add("&Facename");

               // Массив пунктов подменю

          string[] astrText = {"&Times New Roman", "&Arial", "&Courier flew"};
          MenuItem [] ami =new MenuItem[astrText.Length];

          EventHandler ehOnClick = new EventHandler(MenuFacenameOnClick);
          MeasureItemEventHandler ehOnMeasureItem =
                    new MeasureItemEventHandler(MenuFacenameOnMeasureItem);
          DrawItemEventHandler ehOnDrawItem =
                    new DrawItemEventHandler(MenuFacenameOnDrawItem);

          for (int i = 0; i < ami.Length; i++)
          {
               ami[i]              = new MenuItem(astrText[i]);
               ami[i].OwnerDraw    = true;
               ami[i].RadioCheck   = true;
               ami[i].Click       += ehOnClick;
               ami[i].MeasureItem += ehOnMeasureItem;
               ami[i].DrawItem    += ehOnDrawItem;
          }
          miFacename * ami[0];
          raiFacena»e.Checked = true;

          Menu.MenuItems[0].MenuItems.AddRange(ami);
     }
     void MenuFacenameOnClick(object obj, EventArgs ea)
     {
```

см. след. стр.

```
        miFacename.Checked * false;
        miFacename = (MenuItem) obj;
        miFacename.Checked = true;

        Invalidate();
    }
    void MenuFacenameOnMeasureItem(object obj, MeasureItemEventArgs miea)
    {
        MenuItem mi - (MenuItem) obj;
        Foot font = new Font(mi.Text.Substring(1), iFontPointSize);

        StringFormat strfmt * new StringFormat();
        strfmt.HotkeyPrefix = HotkeyPrefix.Show;

        SizeF sizef = miea.Graphics.MeasureString(mi.Text, font,
                                                  1000, strfmt);

        miea.ItemWidth  = (int) Math.Ceiling(sizef.Width);
        miea.ItemHeight = (int) Math.Ceiling(sizef.Height);

        miea.ItemWidth += SystemInformation.MenuCheckSize.Width *
                          miea.ItemHeight /
                              SystemInformation.MenuCheckSize.Height;

        miea.ItemWidth -= SystemInformation.MenuCheckSize.Width;
    }
    void MenuFacenameOnDrawItem(object obj, DrawItemEventArgs diea)
    {
        MenuItem mi    = (MenuItem) obj;
        Graphics grfx = diea.Graphics;
        Brush    brush;

            // Создать Font и StringFormat.

        Font font = new Font(mi.Text.Substring(1), iFontPointSize);
        StringFormat strfmt = new StringFormat();
        strfmt.HotkeyPrefix = HotkeyPrefix.Show;

            // Вычислить размеры отметки и текста.

        Rectangle rectCheck = diea.Bounds;

        rectCheck.Width = SystemInformation.MenuCheckSize.Width *
                          rectCheck.Height /
                              SystemInformation.MenuCheckSize.Height;

        Rectangle rectText = diea.Bounds;

        rectText.X += rectCheck.Width;
```

см. след. стр.

```
            // Нарисовать все.

            diea.DrawBackground();

            if ((diea.State & DrawItemState.Checked) != 0)
                ControlPaint.DrawMenuGlyph(grfx, rectCheck, MenuGlyph.Bullet);

            if ((diea.State & DrawItemState.Selected) != 0)
                brush = SystemBrushes.HighlightText;
            else
                brush = SystemBrushes.FromSystemColor(SystemColors.MenuText);

            grfx.DrawString(mi.Text, font, brush, rectText, strfmt);
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics grfx = pea.Graphics;
            Font font = new Font(miFacename.Text.Substring(1), 12);
            StringFormat strfmt = new StringFormat();
            strfmt.Alignment * strfmt.LineAlignment - StringAlignment.Center;

            grfx.DrawString(Text, font, new SolidBrush(ForeColor), 0, 0);
        }
    }
```

Я задаю 18 полю *iFontPointSize*, просто чтобы шрифт выглядел неуклюже и можно было удостовериться в том, что измерение и рисование работают правильно.

Метод *MenuFacenameOnMeasureItem* начинает работу с получения объекта *MenuItem*, который нужно измерить, и создания шрифта в зависимости от свойства *Text* этого пункта:

```
MenuItem mi = (MenuItem) obj;
Font font = new Font(mi.Text.Substring(1), iFontPointSize);
```

Метод *Substring* свойства *Text* позволяет пропустить амперсант. Затем метод создает объект *StringFormat*, указывающий на то, что буква после амперсанта будет подчеркнута:

```
StringFormat strfmt = new StringFormat();
strfmt.HotkeyPrefix = HotkeyPrefix.Show;
```

Свойство *Text* пункта меню измеряется на основе новых объектов *Font* и *StringFormat:*

```
SizeF sizef = miea.Graphics.MeasureString(mi.Text, font, 1000, strfmt);
```

Структура *sizef* определяет размер пункта меню без отметки:

```
miea.ItemWidth  = (int) Math.Ceiling(sizef.Width);
miea.ItemHeight = (int) Math.Ceiling(sizef.Height);
```

Но ширину нужно увеличить на ширину отметки после того, как отметка была масштабирована соответственно высоте текста:

```
miea.ItemWidth += SystemInformation.MenuCheckSize.Width *
                  miea.ItemHeight /
                  SystemInformation.MenuCheckSize.Height;
```

А затем уменьшена на нормальную ширину отметки:

```
miea.ItemWidth -= SystemInformation.MenuCheckSize.Width;
```

Метод *MenuFacenameOnDrawItem* точно так же создает объекты *Font* и *StringFormat* и заполняет две структуры *Rectangle* на основе свойства *Bounds* объекта *DrawItemEventArgs*. Первый прямоугольник соответствует положению и размеру отметки:

```
Rectangle rectCheck = diea.Bounds;
rectCheck.Width = SystemInformation.MenuCheckSize.Width *
                  rectCheck.Height /
                  SystemInformation.MenuCheckSize.Height;
```

Второй — размеру и положению строки текста:

```
Rectangle rectText = diea.Bounds;
rectText.X += rectCheck.Width;
```

Далее все просто. Метод *DrawBackground* рисует фон, *DrawMenuGlyph* рисует отметку, а *DrawString* — текст, цвет которого определяется тем, выделен пункт меню или нет. Результат:



Для некоторых простых приложений такая исчерпывающая обработка событий *MeasureItem* и *DrawItem* не нужна. Например, следующая программа загружает 64-пиксельное квадратное растровое изображение из ресурса и использует его как пункт меню.

HelpMenu.cs

```
//-------------------------------------------
// HelpMenu.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
```

см. след. стр.

```
using System.Windows.Forms;

class HelpMenu: Form
{
    Bitmap bmHelp;

    public static void Main()
    {
        Application.Run(new HelpMenu());
    }
    public HelpMenuO
    {
        Text = "Help Menu";

        bmHelp = new Bitmap(GetType(), "HelpMenu.Bighelp.bmp");

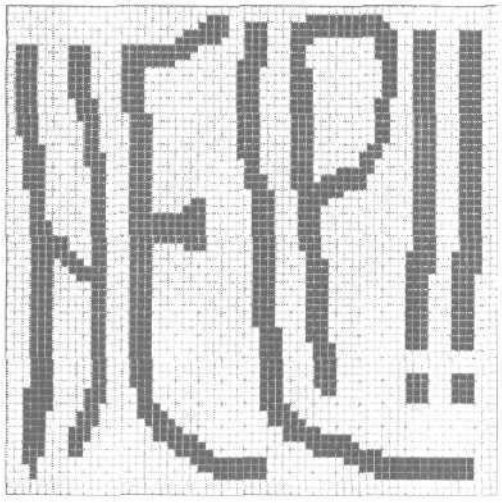
        Menu = new MainMenu();
        Menu.MenuItems.Add("&Help");

        MenuItem mi     = new MenuItem("&Help");
        ffli.QwnerDraw = true;
        mi.Click      += new EventHandler(MenuHelpOnClick);
        mi.DrawItem   += new DrawItemEventHandler(MenuHelpOnDrawItem);
        ffli.MeasureItem  +=
                  new MeasureItemEventHandler(MenuHelpOnMeasureItem);

        Menu.MenuItems[0].MenuItems.Add(mi);
    y
    void MenuHelpOnMeasureItem(object ob], MeasureItemEventArgs miea)
    {
        miea.ItemWidth  * bmHelp.Width;
        miea.ItemHeight = bmHelp.Height;
    }
    void MenuHelpOnDrawItem(object obj, DrawItemEventArgs diea)
    {
        Rectangle rect = diea.Bounds;
        rect.X += diea. Bounds. Width - bmHelp. Width;
        rect.Width – bmHelp.Width; ,

        diea. DrawBackground();
        diea.Graphics.DrawImage(bmHelp, rect);
    }
    void MenuHelpOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("Help not yet implemented.", Text);
    }
}
```

Bighelp.bmp



Обработка *MeasureItem* и *DrawItem* в данной программе весьма проста. Обработчик *MeasureItem* просто устанавливает *ItemWidth* и *ItemHeight* равными ширине и высоте растрового изображения, а *DrawItem* рисует его, по существу выравнивая изображение по правому краю внутри прямоугольника, соответствующего свойству *Bounds*. Похоже, результат отражает отчаяние пользователя-новичка:



ГЛАВА

15



Контуры, области и отсечение

Если вам доводилось заниматься программированием графики в PostScript, вы, вероятно, уже знаете, что такое графический *контур* (path). В PostScript и шагу не сделать без контура, который в этом языке возведен в ранг центрального объекта для рисования. Хотя другие графические среды не дошли до таких крайностей, как PostScript, в настоящее время контур признан ценным инструментом для программирования графики.

Графический контур позволяет очень просто соединять прямые и кривые. Как вы уже знаете, соединенные прямые рисуют методом *DrawLines*, а для рисования соединенных кривых Безье служит метод *DrawBeziers*, но до сих пор ничего не сказано о том, как соединять прямые с кривыми Безье. Именно это делает графический контур. Несмотря на простоту, использование графических контуров открывает доступ к самым разным методикам рисования, изучению которых посвящена эта глава (а также главы 17 и 19).

Контуры можно также использовать для *отсечения* (clipping). Отсечение — это ограничение размеров области, в которой выводится графика, определенной частью экрана или страницы принтера. В действительности контур, которым задан отсеченный участок, сначала преобразуется в *область*. Область описывает зону, в которой устройство выводит графику, в координатах на устройстве.

Задача и ее решение

Начнем изучение графических контуров с задачи из программирования графики. Допустим, требуется нарисовать пером, толщина которого существенно больше 1 пиксела, фигуру, образованную полуокружностью и двумя прямыми, соединенными с ее концами. Вот что получилось в результате первой попытки нарисовать такую фигуру:

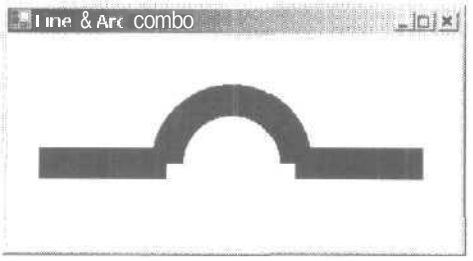
LineArcCombo.cs

```
//-----------------------------------------
// LineArcCombo.cs © 2001 fey Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class LineArcCombo: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new LineArcCombo());
     }
     public LineArcCombo()
     {
          Text = "Line & Arc Combo";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen pen = new Pen(clr, 25);

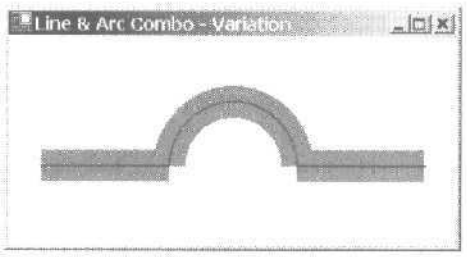
          grfx.DrawLine(pen, 25, 100, 125, 100);
          grfx.DrawArc (pen, 125, 50, 100, 100, -180, 180);
          grfx.DrawLine(pen, 225, 100, 325, 100);
     }
}
```

Программа рисует две линии длиной по 100 единиц (т. е. 100 пикселов в случае дисплея или 1 дюйм в случае принтера) и полуокружность, образующую *дугу* диаметром 100 единиц, толщина пера составляет 25 единиц. А вот как все это выглядит:



Такой результат, может, удовлетворит вас, но не меня. Я бы хотел, чтобы прямые и дуга соединялись. Конечно, эти линии соприкасаются, но они никак не выглядят соединенными, да и выступы на концах дуги здесь тоже ни к чему.

Если изменить программу LineArcCombo и заставить ее рисовать эту фигуру дважды (первый раз толстым пером серого цвета, затем — тонким черным толщиной в 1 пиксел), картина станет яснее:



Оказывается, линии толщиной 25 пикселов просто выдаются на 12 пикселов по сторонам от линий толщиной 1 пиксел. Поскольку и обе прямые, и дуга нарисованы при помощи отдельных вызовов соответствующих методов, каждый элемент фигуры — это отдельная сущность. В двух точках, где встречаются прямые и дуга, толстые линии соприкасаются, но не образуют единое целое.

Быть может, этой фигуре все же удастся придать должный вид, немного поколдовав с координатами. Например, можно опустить дугу на 12 пикселов или что-то в этом роде. Но это, честно говоря, всего лишь косметические меры, но не решение задачи.

Чтобы решить ее, нужно придумать, как намекнуть графической системе, что от нее требуется соединить прямые с дугой. Если бы мы имели дело только с прямыми, задача решалась бы элементарно: вместо рисования отдельных линий методом *DrawLine* можно было бы нарисовать составную линию при помощи *DrawLines*. Вот, к примеру, программа, рисующая нечто похожее на то, что нам нужно,

WidePolyline.cs

```
//---------------------------------------
// WidePolyline.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class WidePolyline: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new WidePolyline());
     }
     public WidePolyline()
     {

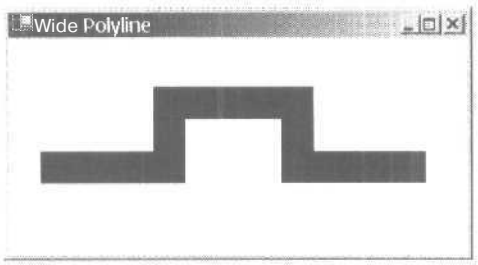
          Text = "Wide Polyline";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen pen = new Pen(clr, 25);

          grfx.DrawLines(pen, new Point[] {
```

см. след. стр.

```
                    new Point ( 25,  100), new Point(125,  100),
                    new Point(125,   50), new Point(225,   50),
                    new Point(225,  1GS), new Point(325,  100) });
    }
}
```

При вызове метода *DrawLines* создается массив из шести структур *Point*, определяющий ломаную линию из пяти отрезков:



Графическая система «знает», что пользователь хотел бы соединить эти линии, поскольку все они заданы в одном вызове функции, поэтому в местах соединения отрезков толстая линия нарисована корректно.

Удачное применение составной линии в WidePolyline наводит на мысль о существовании другого решения нашей задачи рисования фигуры, содержащей дугу. Заглянув в главу 5, можно вспомнить, как рисуют эллипс при помощи составной линии, и нарисовать дугу именно так. Другой вариант — преобразовать прямые в кривые Безье (для этого нужно расположить управляющие точки кривых между конечными и на одной прямой с ними); преобразовать дугу в одну или несколько кривых Безье (при помощи формул из главы 13) и нарисовать всю фигуру целиком, вызвав метод *DrawBeziers*.

Но должен же быть менее замысловатый способ, позволяющий указать графической системе, что нужно соединить концы дуги с прямыми! Здесь требуется метод вроде *DrawLines,* который работал бы с комбинациями прямых и кривых. Раз уж мы взялись за это, можно пожелать, чтобы наша «волшебная функция» заодно работала с кривыми Безье и фундаментальными сплайнами.

Такой «волшебной функцией» (точнее, волшебным классом) является *GraphicsPath*. Вот программа LineArcPath, которая корректно рисует нашу фигуру. Она всего на три оператора длиннее LineArcCombo.

LineArcPath.cs

```
//---------------------------------------
// LineArcPath.cs © 2001 by Charles Petzold
//_____,__ -- __ _ _ ,_____.
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Winows.Forms;
```

см. след. стр.

```
class LineArcPath: PrintableForm
{
    public new static void Main()
    {
        Application.ftun(newLineArcPath());
    }
    public LineArcPath()
    {
        Text - "Line & Arc in Path";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    i
        GraphicsPath path = new GraphicsPath();
        Pen          pen  = new Pen(clr, 25);

        path.AddLine( 25, 100, 125, 100);
        path.AddArc £125,  50, 100, 100, -180, 180);
        path.AddLine(225, 100, 325, 100);

        grfx.DrawPath(pen, path);
    }
}
```

Первый из трех дополнительных операторов размещен в начале метода *DoPage* и создает контур:

```
GraphicsPath path = new GraphicsPath();
```

Хотя класс, в котором реализованы контуры, называется *GraphicsPath,* я назвал переменную, содержащую экземпляры этого класса, просто *path*. Класс *GraphicsPath* определяется в пространстве имен *System.Drawing.Drawing2D,* а дополнительный оператор *using* нужен другим операторам, добавленным к этой программе.

LineArcCombo рисует первую прямую, используя метод *DrawLine* из класса *Graphics:*

```
grfx.DrawLine(pen, 25, 100, 125, 100);
```

В LineArcPath вместо него применяется метод *AddLine* класса *GraphicsPath:*

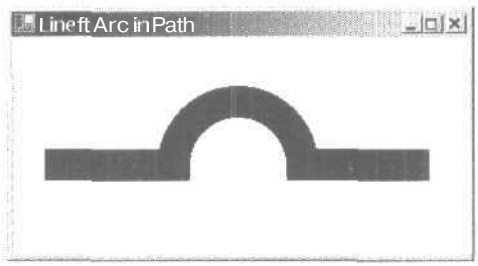
```
path.AddLine(25, 100, 125, 100);
```

У *AddLine* нет аргумента *Pen,* но остальные аргументы идентичны таковым метода *DrawLine.* Метод *AddArc* также отличается от *DrawArc* отсутствием аргумента *Pen. AddLine* и *AddArc* ничего не рисуют: координаты, заданные при их вызове, просто накапливаются, составляя контур.

Наконец, контур появляется на экране (его рисует третий из добавленных к этой программе операторов):

```
grfx.DrawPath(pen, path);
```

Заметьте *DrawPath* — это метод класса *Graphics,* нашего старого знакомого. В результате вызова *DrawPath* получается именно такая фигура, которая нам ну» на:



Разработчики с опытом программирования Win32 API или MFC могут заметить, что реализация графических контуров в Windows Forms концептуально иная. В Win32 API функция *BeginPath* переводит контекст устройства в специальный режим, в котором вызовы обычных графических функций (*LineTo*, *BezierTo* и т. п,) не визуализируются, а становятся частью контура. Формирование контура завершается вызовом *EndPath*, после чего контур можно нарисовать, вызвав *StrokePath* (или сделать с ним что-то еще).

Подход, используемый в Windows Forms, намного гибче. Win32 API допускает существование лишь одного контура в контексте некоторого устройства, *а* в Windows Forms можно создавать и хранить сколько угодно контуров. Кроме того, объект *Graphics* не нужен для создания контура. Контур существует независимо от объектов *Graphics*, пока он не будет визуализирован методом *DrawPath* (или пока над контуром не выполнят иное действие),

Фактически можно изменить программу LineArcPath, чтобы она сохраняла объект *GraphicsPath* как поле. Можно создавать контур и вызывать методы *AddLine* и *DrawPath* в конструкторе формы, после чего методу *DoPage* останется просто создать перо и вызвать *DrawPath*. Если же действительно нужно покончить со всеми приготовлениями в конструкторе, можно сделать оператор для создания пера полем формы.

Более формально о контурах

Теперь рассмотрим контур более формально и приведем его краткий обзор. Для начала дадим несколько определений:

Контур (path) — это независимый от устройства набор точек координат, описывающий совокупность прямых и кривых. Эти прямые и кривые могут быть как соединенными, так и независимыми. Любой набор соединенных прямых и кривых в составе контура называется *фигурой* или *субконтуром* (оба термина используются в интерфейсе Windows Forms и являются синонимами). Таким образом, контур — это совокупность фигур, число которых может равняться 0, а любая фигура — это совокупность соединенных прямых и кривых. Контур, созданный программой LineArcPath, состоит всего лишь из одной фигуры.

Фигура может быть *открытой* или *закрытой*. Фигура является закрытой, если конец ее последней линии соединен с началом первой (для закрытия фигур служит специальный метод класса *GraphicsPath* — *CloseFigure*), иначе фигура считается открытой. Единственная фигура, созданная LineArcPath, является открытой.

Я уже демонстрировал метод *DrawPath,* рисующий на устройстве вывода прямые и кривые, составляющие контур. Класс *Graphics* также включает метод *FillPath*,

который при помощи кисти заливает внутренние области всех закрытых фигур контура. При исполнении метод *FillPath* закрывает *все* открытые фигуры контура, чтобы превратить их в замкнутые области.

Как я покажу ниже, можно также преобразовать контур в *область*. В отличие от контура (который является набором прямых и кривых) область описывает часть поверхности экрана, которая может быть как простой (например, прямоугольной), так и довольно сложной формы. Область можно применять и для отсечения. Отсечение ограничивает рисование заданной областью поверхности дисплея.

Программисты, впервые встретившись с контурами, склонны думать что контур — это нечто существенно большее, чем просто совокупность определений прямых и кривых. Чтобы избавиться от этой иллюзии, рассмотрим свойства объекта *GraphicsPath*. В контуре нет постоянных данных, кроме тех, что доступны через свойства этого объекта:

Свойства *GraphicsPath*

Тип	Свойство	Доступ	Описание
FillMode	*FillMode*	Чтение/запись	*FillMode.Alternate* или *FillMode.Winding*
int	*PointCount*	Чтение	Число точек контура
PointF[]	*PathPoints*	Чтение	Массив точек координат
byte[]	*PathTypes*	Чтение	Массив типов точек
PathData	*PathData*	Чтение	То же, что *PathPoints* или *PathTypes*

Перечисление *FillMode* также используется методами *DrawPolygon* и *DrawClosedCurve*, с которым мы познакомились в главах 5 и 13. В случае контура свойство *FillMode* определяет способ его заливки (т. е. способ преобразования в область), если контур содержит пересекающиеся линии.

Остальные свойства несут избыточную информацию, определяя всего два идентичных по размеру массива:

- массив *PathPoints* структур *PointF*;
- массив *PathTypes* значений типа *byte*.

Свойство *PointCount* предоставляет число элементов этих массивов (его также можно получить из *PathPoints.Length* или *PathTypes.Length*).

Свойство *PathData*, представляющее собой объект типа *PathData*, определяемый в пространстве имен *System.Drawing.Drawing2D*, обеспечивает дополнительный уровень избыточности. У класса *PathData* два свойства:

Свойства *PathData*

Тип	Свойство	Доступ	Описание
PointF[]	*Points*	Чтение/запись	Массив точек координат
byte[]	*Types*	Чтение/запись	Массив типов точек

Для любого объекта *GraphicsPath* массив *Points*, содержащийся в *свойстве PathData*, идентичен значению свойства *PathPoints*, а массив *Types* из свойства *PathData* — значению свойства *PathTypes*.

Значения типа *byte*, составляющие массив из свойства *PathTypes*, реально представляют собой значения перечисления *PathPointType*, также определяемого в пространстве имен *System.Drawing.Drawing2D*.

Перечисление ***PathPointType***

Член	Значение
Start	0
Line	1
Bezier или *Bezier3*	3
PathTypeMask	7
DashMode	16
PathMarker	32
CloseSubpath	128

С каждой структурой *PointF* из массива *PathPoints* связано значение *PathPointType*. Это может быть *Start*, *Line* или *Bezier*. Тип *Start* идентифицирует первую точку фигуры, *Line* — точку, определяющую прямую, а *Bezier* — точку кривой Безье. По мере добавления к контуру дуги и канонические сплайны преобразуются в кривые Безье. Думаю, после того как в главе 13 я показал точность аппроксимации окружности кривыми Безье, никто не усомнится в допустимости подобных преобразований.

Последние три значения перечисления *PathPointType* — это флаги, которые можно комбинировать со значениями *Start, Line* или *Bezier*. Как я покажу ниже, флаги *PathMarker* и *CloseSubpath* генерируются при вызове метода *GraphicsPath*.

Член *PathTypeMask* — это битовая маска, позволяющая отделять в значениях массива *Types* типы точек *(Start, Line* или *Bezier)* от флагов (*DashMode*, *PathMarker* или *CloseSubpath*).

Контур *не содержит* ничего такого, что связывало бы координаты составляющих его точек с размерами в реальном мире. Вопрос о том, в каких единицах исчисляются координаты точек контура (пикселах, дюймах, миллиметрах или каких-либо еще), не имеет смысла — это просто точки. Их координаты преобразуются в пикселы, дюймы или миллиметры лишь при визуализации контура на устройстве вывода,

Создание контура

У класса *GrapbicsPath* шесть конструкторов:

Конструкторы *GraphicsPath*

```
GraphicsPath()
GraphicsPath(Point[] apt, byte[] abyPotntType)
GraphicsPath(PointF[] aptf, byte[] abyPointType)
GraphicsPath(FillMode fm)
GraphicsPath(Point[] apt, byte[] abyPointType, FillMode fm)
GraphicsPath(PointF[] aptf, byte[] abyPointType, FillMode fm)
```

Если аргумент *FillMode* не задан, по умолчанию принимается значение *FillMode.Alternate*.

Как видно из определения четырех из этих конструкторов, контур может быть создан при помощи массива структур *Point* или *PointF* и набора соответствующих

значений перечисления *PathPointType* заданных в виде массива значений типа *byte*. Но маловероятно, что программа начнет создавать контур таким способом. Вместо этого конструкторы наиболее эффективно используются для изменения значений *PathPoints* существующего контура.

Обычно новый контур создают при помощи конструктора по умолчанию:

```
GraphicsPath path = new GraphicsPath();
```

Затем можно вызывать методы класса *GraphicsPath*, чтобы добавлять к контуру прямые или кривые. Эти методы аналогичны соответствующим методам класса *Graphics* за исключением того, что их имена начинаются с префикса *Add* вместо *Draw* и у них нет аргумента *Pen*.

Вот методы объекта *GraphicsPath*, добавляющие к текущей фигуре прямые, кривые Безье, дуги и канонические сплайны. Ниже аргументы методов не показаны, так как они в основном те же, что у соответствующих методов с префиксом *Draw*, определенных в классе *Graphics*:

Методы *GraphicsPath* (выборочно)

```
void AddLine(...)
void AddLines(...)
void AddArc(...)
void AddBezier(...)
void AddBeziers(...)
void AddCurve(...)
```

По мере добавления к контуру дуги и канонические сплайны преобразуются в кривые Безье.

Если *path* — это объект типа *GraphicsPath*, следующие три вызова добавят к контуру три соединенных линии:

```
path.AddLine(0, 0, 0, 100);
path.AddLine(0, 100, 100, 100);
path.AddLine(100, 100, 100, 0);
```

В результате будет нарисован квадрат без верхней стороны. Я выбрал такие координаты, чтобы конечная точка одной линии совпадала с начальной точкой другой, словно я рисую эту фигуру обычным способом.

Но при определении контура быть столь скрупулезным вовсе не обязательно. Если не задано обратное (как — покажу ниже), то все линии, дуги, кривые Безье и канонические сплайны, добавляемые к контуру, в конечном счете становятся частью одной и той же фигуры. Если координаты конечной точки последнего элемента не совпадают с координатами начальной точки следующего, контур автоматически генерирует прямую, соединяющую эти точки. Если вообще убрать второй из трех показанных выше операторов, результат не изменится:

```
path.AddLine(0, 0, 0, 100);
path.AddLinedOO, 100, 100, 0);
```

Поскольку первая линия заканчивается в точке с координатами (0, 100), а вторая начинается в точке (100, 100), контур добавляет линию, соединяющую эти две точки.

Можно использовать и следующие три метода:

Методы *GraphicsPath*(выборочно)

```
void StartFigure()
void CloseFigureO
void CloseAllFigures()
```

Каждый из них завершает текущую фигуру и начинает новую, а *CloseFigure* еще и закрывает текущую фигуру. При необходимости к контуру автоматически добавляется прямая, соединяющая первую и последнюю точки завершаемой фигуры. Метод *CloseAllFigures* закрывает все фигуры, составляющие контур в данный момент,

вызовы:

```
path.AddLine(0, O, O, 100);
path.AddLine(0, 100, 100, 100);
path.AddLine(100, 100. 100, 0);
path.AddLine(100, 0, O, O);
path.CloseFigure();
```

явно создают закрытую фигуру квадратной формы. Вызовы;

```
path.AddLine(0, O, O, 100);
path.AddLinedOO, 100, 100, 0);
path.CloseFigure();
```

создают ту же самую закрытую фигуру, вынуждая контур автоматически добавить линии, образующие верхнюю и нижнюю стороны квадрата. Вызовы:

```
path.AddLine(0, O, O, 100);
path.AddLine(0, 100, 100, 100);
path.AddLinedOO. 100, 100, 0);
path.AddLinedOO, 0, O, O);
path.StartFigure();
```

тоже создают фигуру в виде квадрата с полным набором сторон, но эта фигура не считается закрытой, поскольку контур не заканчивается вызовом *CloseFigure*.

Следующие методы начинают новую фигуру, которая затем закрывается:

Методы *GraphicsPath* (выборочно)

```
void AddRectangle(...)
void AddRectangles(...)
void AddPolygon(...)
void AddEllipse(...)
void AddPie(...)
void AddClosedCurve(...)
```

Например, вызовы:

```
path.AddLineCO, 0, 100, 0);
path.AddRectangle(new Rectangle(50, 50, 100, 100);
path.AddLine(200, 0, 0, 0);
```

создают три фигуры, состоящие из:

- одной линии (открытая фигура);
- четырех линий (закрытая фигура);
- одной линии (открытая фигура).

Контуры также можно добавлять друг к другу:

Метод *AddPath* класса *GraphicsPath*

```
void AddPath(GraphicsPath path, bool bConnect)
```

Второй аргумент этого метода показывает, должен ли добавляемый контур быть соединен с текущей фигурой.

Методы *AddString* добавляют к контуру текстовую строку. Синтаксис этих методов заметно отличен от синтаксиса методов *DrawString*:

Методы *AddString* класса *GraphicsPath*

```
void AddString(string str, FontFamily ff, int iStyle, float fSize,
               Point pt, StringFormat sf)
void AddString(string str, FontFamily ff, int iStyle, float fSize,
               PointF ptf, StringFormat sf)
void AddStringCstring str, FontFamily ff, int iStyle, float fSize,
               Rectangle rect, StringFormat sf)
void AddString(string str, FontFamily ff, int iStyle, float fSize,
               RectangleF rectf, StringFormat sf)
```

Несмотря на наличие аргументов, совершенно не похожих на точки координат, эти методы делают не что иное, как добавляют к контуру контуры символов шрифта в виде наборов прямых и кривых Безье.

На самом деле аргументы метода *AddString* не столь эксцентричны, как можно предположить из определения метода. Третий аргумент определяется как значение типа *int,* но реально это член перечисления *FontStyle (Regular, Bold, Italic, Underline* или *Strikeout*). Второй, третий и четвертый аргументы те же, что у конструктора объекта *Font.*

Но почему бы методам *AddString* не использовать аргументы типа *Font,* как это делают методы *DrawString?* Дело в том, что для объекта *Font* обычно уже задан определенный размер символов шрифта, а контур не сохраняет сведений о размерах. Аргумент *fSize* метода *AddString* задает вовсе не размер символов шрифта в пунктах. Определение аргумента *fSize* у метода *AddString* аналогично созданию объекта *Font* с размером, выраженном в пикселах, и аргументами *GraphicsUnit.Pixel* или *GraphicsUnit.World* (см. главу 9). До визуализации текст не содержит сведений о своих размерах.

Возможность размещения в контуре текста открывает доступ к такому богатому разнообразию эффектов, что я посвятил им отдельную главу «Игры со шрифтами».

В контур также можно вставлять маркеры, выполняющие функции закладок:

Методы *GraphicsPath* (выборочно)

```
void SetMarkers()
void ClearMarkers()
```

Позже можно выполнять поиск этих маркеров при помощи класса *GraphicsPathIterator*. Это средство облегчает редактирование контура,

Визуализация контура

Чаще всего контур визуализируют при помощи одного из методов класса *Graphics*:

Методы *Graphics* (выборочно)

```
void DrawPath(Pen pen, GraphicsPath path)
void FillPath(Brush brush, GraphicsPath path)
```

Метод *DrawPath* рисует заданным пером прямые и кривые, составляющие контур. *FillPath* заливает внутренние области всех замкнутых фигур, используя заданную кисть. Хотя этот метод закрывает открытые фигуры, чтобы выполнить заливку, он не вносит в контур постоянных изменений. Если в контуре есть пересекающиеся линии, то заливка внутренних областей определяется значением свойства *FillPath* объекта *GraphicsPath*. При визуализации все точки контура подвергаются любым преобразованиям, установленным для объекта *Graphics*.

А теперь посмотрим, как все это работает на практике. Программа Flower рисует цветок при помощи контуров и преобразования.

Flower.cs

```
//---------------------------------
// Flower.cs © 2001 by Charles Petzold
//---------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class Flower: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new Flower());
     }
     public FlowerO
     {
          Text = "Flower";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
               // Нарисовать зеленый стебель, протянувшийся из нижнего
               // левого угла к центру.

          grfx.DrawBezier(new Pen(Color.Green, 10),
                    new Point(0, cy),          new Point(0, 3 * cy / 4),
                    new Point(cx / 4, cy / 4), new Point(cx / 2, cy / 2));
```

см. след. стр.

```
            // Задать преобразование дня остальных частей цветка,

        float fScale = Math.Min(cx, cy) / 2000f;
        grfx.TranslateTransform(cx / 2, cy / 2);
        grfx.ScaleTransform(fScale, fScale);

            // Нарисовать красные лепестки.

        GraphicsPath path - new GraphicsPath();

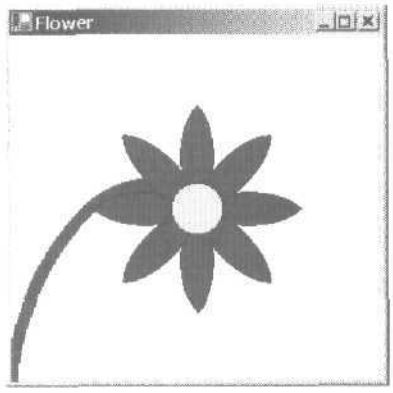
        path.AddBezier(new Point( 0,    0), new Point(125,  125),
                       new Point(475,  125), new Point(600,    0));
        path.AddBezier(new Point(600,    0), new Point(475, -125),
                       new Point(125, -125), new Point( 0,    0));

        for (int i = 0; i < 8; i++)
        {
            grfx.FillPath(Brushes.Red, path);
            grfx.DrawPath(Pens.Black, path);
            grfx.RotateTransform(360 / 8);
        }

            : // Нарисовать желтый круг в центре.

        Rectangle rect = new Rectangle(-150, -150, 300, 300);
        grfx.FillEllipse(Brushes.Yellow, rect);
        grfx.DrawEllipse(Pens.Black, rect);
    }
}
```

Исполнение метода *DoPage* начинается с рисования изображающей стебель цветка кривой Безье, которая начинается из левого нижнего угла и заканчивается в центре клиентской области (или страницы принтера). Далее программа задает глобальное преобразование, чтобы создать изотропную область для рисования, разбитую на четыре квадранта. Координаты этой области меняются от -1000 до 1000, а начало координат расположено в центре области.



Далее программа должна нарисовать несколько лепестков, и здесь в дело вступает контур. Если бы лепестки были эллиптическими, я просто вызвал бы метод *FillEllipse*. Но форма лепестка точнее определяется парой кривых Безье, а для заливки полученной фигуры нужен контур. Создав контур, программа восемь раз вызывает методы *FillPath* и *DrawPath*. Каждый раз после их вызова *RotateTransform* изменяет глобальное преобразование объекта *Graphics*, в результате лепестки располагаются по кругу вокруг центра цветка. Метод *DoPage* завершается рисованием желтого круга в центре клиентской области.

Уверен, вы помните программу Scribble из главы 8. Тогда я показывал, как сохранить все нарисованные пользователем линии при помощи класса *ArrayList*. Этот объект напоминает массив, способный динамически изменять свой размер. На самом деле применение объекта *ArrayList* весьма напоминает сохранение координат в контуре. Замена *ArrayList* объектом *GrapbicsPatb* позволяет существенно упростить программу, которая в этом случае становится даже проще, чем версия из главы 11 (ScribbleWithBitmap), сохраняющая рисунок при помощи теневого изображения.

ScribbleWifhPath.cs

```
//---------------------------------------------
// ScribbleWithPath.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ScribbleWithPath: Form
{
     GraphicsPath path;
     bool         bTracking;
     Point        ptLast;

     public satic void Main()
     {
          Application.Run(new ScribbleWithPath());
     }
     public ScribbleWithPath()
     {
          Text = "Scribble with Path";
          BackColor - SystemColors.Window;
          ForeColor = SystemColors.WindowText;

               // Создать контур.

          path - new GraphicsPath();
     }
     protected override void OnMouseDown(MouseEventArgs mea)
```

см. след. стр.

```
    {
        if (mea.Button != MouseButtons.Left)
            return;

        ptLast = new Point(mea.X, mea.Y);
        bTracking = true;

            // Начать фигуру.

        path.StartFigure();
    }
    protected override void OnMouseMove(MouseEventArgs mea)
    {
        if (!bTracking)
            return;

        Point ptNew = new Point(mea.X, mea.Y);

        Graphics grfx = CreateGraphics();
        grfx.DrawLine(new Pen(ForeColor), ptLast, ptNew);
        grfx.Dispose();

            // Добавить линию.

        path.AddLine(ptLast, ptNew);

        ptLast = ptNew;
    }
    protected override void OnMouseUp(MouseEventArgs mea)
    {
        bTracking = false;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
            // Нарисовать контур.

        pea.Graphics.DrawPath(new Pen(ForeColor), path);
    }
}
```

Помимо дополнительного оператора *using,* для превращения версии программы Scribble, не сохраняющей пользовательские рисунки, в ScribbleWithPath, нужно определить контур как переменную-поле и добавить всего 4 оператора, прокомментированные в тексте программы.

Контур создается в конструкторе формы. Когда курсор находится над клиентской областью формы, нажатие любой кнопки мыши вызывает метод *StartFigure*, начинающий новую фигуру. Вызов *AddLine* во время исполнения метода *OnMouseMove* добавляет к контуру новую линию, а метод *OnPaint* просто вызывает *DrawPath*,

Преобразования контуров

В классе *GraphicsPatb* есть несколько методов, позволяющих программе модифицировать контур. Первый из них может легко запутать программиста (по крайней мере со мной так и было, когда я столкнулся с этим методом впервые).

Метод *Transform* класса *GraphicsPath*

```
void Transform(Matrix matrix)
```

Как сказано в главе 7, у класса *Graphics* есть свойство *Transform* типа *Matrix*. Свойство *Transform* объекта *Graphics* влияет на всю графику, которая будет выводиться после его определения.

Но *Transform* из класса *GraphicsPath* — это совсем другое дело. В классе *GraphicsPath* это не свойство объекта *GraphicsPatb*. а метод, и это очень важно. Свойство — это, как правило, характеристика объекта, а метод обычно выполняет какое-либо действие. Иначе говоря, свойство можно сравнить с прилагательным, а метод — с глаголом.

Метод *Transform* из класса *GraphicsPath* изменяет координаты контура, применяя к ним заданное преобразование, эффект которого становится постоянным. Метод *Transform* не влияет на координаты точек, добавленных к контуру после вызова этого метода. Кроме того, объект *GraphicsPath* не сохраняет сведений о преобразовании. Так, если задан объект *matrix* типа *Matrix*, описывающий удвоение координат всех точек, то при вызове:

```
grfx.Transform(matrix);
```

результат будет тот же, что при рисовании контура на основе массива модифицированных точек, полученного путем удвоения координат точек из массива *PathPoints*.

Transform — единственный метод в классе *GraphicsPath,* имеющий отношение к матричным преобразованиям. Чтобы задействовать его, надо воспользоваться классом *Matrix,* определяемым в пространстве имен *System.Drawing.Drawing2D*. С этим классом мы встречались в конце главы 7. Проще всего применить класс *Matrix,* создав сначала единичную матрицу конструктором по умолчанию:

```
Matrix matrix = new Matrix();
```

Затем это преобразование можно изменить, используя различные методы класса *Matrix. Translate* — это практически полная копия метода *TranslateTransform* класса *Graphics.* (Не удивительно, если на самом деле метод *TranslateTransform* объекта *Graphics* реализован просто вызовом соответствующего метода *Translate* свойства *Transform* этого объекта.)

Методы ***Translate* класса *Matrix***

```
void Translate (float dx, float dy)
void Translate (float dx, float dy, MatrixOrder mo)
```

Перечисление *MatrixOrder* состоит из двух членов: *Append* и *Prepend*.

Метод *Scale* очень похож на метод *ScaleTransform* из класса *Graphics*:

Методы *Scale* класса *Matrix*

```
void Scale(float sx, float sy)
void Scale(float sx, float sy, MatrixOrder mo)
```

Выше я привел пример с удвоением координат точек контура. То же позволяет сделать код:

```
Matrix matrix = new Matrix();
matrix.Scale(2, 2);
path.Transform(matrix);
```

В класс *Matrix* также входит метод *Rotate*:

Методы *Rotate* класса *Matrix*

```
void Rotate(float fAngle)
void Rotate(float fAngle, MatrixOrder mo)
```

Можно изменить программу Flower, чтобы вместо метода *RotateTransform* из класса *Graphics* она использовала метод *Rotate* из класса *Matrix*. В этом случае после создания контура следует создать объект *Matrix,* описывающий поворот на 45°:

```
Matrix matrix = new Matrix();
matrix.Rotate(45);
```

Далее вместо *RotateTransform* цикл *for* будет вызывать метод *Transform* контура:

```
path.Transform(matrix);
```

В исходной версии программы Flower координаты контура не изменяются, а метод *RotateTransform* меняет способ преобразования координат во время визуализации контура при помощи объекта *Graphics*. Модифицированная же версия программы выполняет поворот, изменяя значения координат, хранящиеся в контуре. К завершению цикла *for* (после восьми поворотов на 45°) восстанавливаются исходные значения координат контура.

Вот интересные методы класса *Matrix*, у которых нет двойников в классе *Graphics*:

Методы *RotateAt* класса *Matrix*

```
void RotateAt(float fAngle, PointF ptf)
void RotateAt(float fAngle, PointF ptf, MatrixOrder mo)
```

Обычно матричное преобразование вращает изображение вокруг точки с координатами (0, 0). Этот метод позволяет указать точку, вокруг которой осуществляется поворот. Пусть создается контур:

```
GraphicsPath path = new GraphicsPath();
path.AddRectangle(new Rectangle(0, 0, 100, 100));
```

с точками (0, 0), (100, 0), (100, 100) и (0, 100). Если после этого создать объект *Matrix,* вызвать метод *Rotate* для поворота на 45° и применить к контуру преобразование:

```
Matrix matrix = new Matrix();
matrix.Rotate(45);
path.Transform(matrix);
```

координаты контура примут значения (округленно) (0, 0), (70.7, 70.7), (0. 141.4) и (-70, 70). Если вместо этого вызвать метод *RotateAt*, задающий вращение вокруг центра прямоугольника:

```
Matrix matrix = new MatrixO;
matrix.RotateAt(45, new Point(50, 50));
path.Transform(matrix);
```

координаты точек контура примут значения (50, -20.7), (120.7, 50), (50, 120.7) и (-20.7, 50).

В классе *Matrix* тоже есть методы, выполняющие сдвиг:

Методы *Shear* класса *Matrix*

```
void Shear(float xShear, float yShear)
void Shear(float xShear, float yShear, MatrixOrder mo)
```

Результат применения этого метода к преобразованию по умолчанию можно описать формулами:

$$x' = x + xShear \cdot y$$
$$y' = yShear \cdot x + y$$

Другие виды модификации контура

Transform— не единственный метод класса *GraphicsPath*, модифицирующий все координаты контура. Метод *Flatten* предназначен для преобразования всех кривых Безье в контуре в прямолинейные сегменты:

Методы *Flatten* класса *GraphicsPath*

```
void Flatten()
void Flatten(Matrix matrix)
void Flatten(Matrix matrix, float fFlatness)
```

Также есть возможность перед выравниванием преобразовать координаты точек при помощи объекта *Matrix*.

По мере увеличения значения аргумента *fFlatness* число сегментов линии уменьшается. По умолчанию для этого аргумента задано значение 0,25. Метод *Flatten* не определен для значения 0.

Метод *Widen* влияет на контур намного сильнее, чем *Flatten*. Первым аргументом этого метода всегда является объект *Pen*:

Методы Widen класса *GraphicsPath*

```
void Widen(Pen pen)
void Widen(Pen pen, Matrix matrix)
void Widen(Pen pen, Matrix matrix, float fFlatness)
```

Этот метод игнорирует цвет пера и использует только его толщину, которая обычно не меньше двух единиц. Представьте себе тот же контур, но нарисованный толстым пером. Так вот, новый контур будет представлять собой контуры этой толстой линии. Каждый открытый контур преобразуется в закрытый, а каждый закрытый — в пару закрытых контуров. Перед утолщением контура метод *Widen* преобразует все кривые Безье в составные линии. Для этого преобразования также можно задать коэффициент сглаживания или преобразовать координаты контура перед его утолщением при помощи объекта *Matrix*.

Порой результаты метода *Widen* выглядят странно, поэтому полезно взглянуть на следующую программу: в конструкторе создается контур, состоящий из открытой V-образной фигуры и одной закрытой фигуры в форме треугольника,

WidenPath.cs

```
//-------------------------------------
// WidenPath.cs © 2001 by Charles Petzold
// ------------ ..... ----------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class WidenPath: PrintableForm
{
     GraphicsPath path;

     public new static void Main()
     {
          Application.Run(new WidenPath());
     }
     public WidenPath()
     {
          Text = "Widen Path";

          path = new GraphicsPath();

               // Создать открытую фигуру.

          path.AddLines(new Point[] { new Point(20, 10),
                                      new Point(50, 50),
                                      new Point(80, 10) });

               // Создать закрытую фигуру.

          path.AddPolygon(new Point[] { new Point(20, 30),
                                        new Point(50, 70),
                                        new Point(80, 30) });
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
```

См. след. стр.

```
    {
        grfx.ScaleTransform(cx / 300f, cy / 200f);

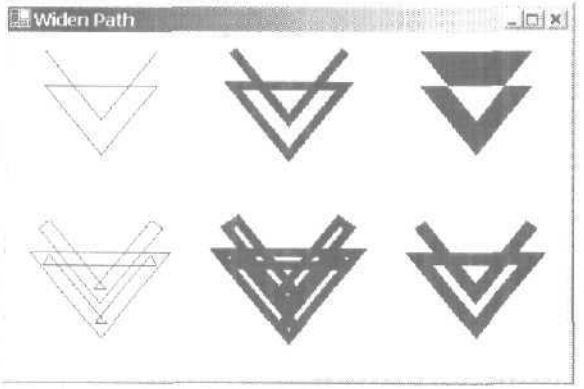
        for (int i = 0; i < 6; i++)
        i
            GraphicsPathpathClone= (GraphicsPath) path.Clone();
            Matrix       matrix    = new Matrix();
            Pen          penThin   = new Pen(clr, 1);
            Pen          penThick  = new Pen(clr, 5);
            Pen          penWiden  = new Pen(clr, 7.5f);
            Brush        brush     = new SolidBrush(clr);

            matrix.Translate((i % 3) * 100, (i / 3) * 100);

            if (i < 3)
                pathClone.Transform(matrix);
            else
                pathClone.Widen(penWiden, matrix);

            switch (i % 3)
            {
            case 0:  grfx.DrawPath(penThin, pathClone);    break;
            case 1:  grfx.DrawPath(penThick, pathClone);   break;
            case 2:  grfx.FillPath(brush, pathClone);      break;
            }
        }
    }
}
```

Метод *DoPage* создает *6* копий контура при помощи метода *Clone,* а затем вызывает метод *Transform,* чтобы поместить каждую копию на свое место. После этого программа прорисовывает контур 6 разными способами. Вот результат:



В верхнем ряду показаны контуры, нарисованные пером шириной в 1 единицу, 5 единиц и залитый контур. Контуры в нижнем ряду нарисованы так же, как соответствующие фигуры из верхнего ряда, но после этого был вызван метод *Widen* с пером шириной в 7,5 единиц.

Эффект метода *Widen* отчетливее всего виден на картинках слева. Открытая V-образная фигура становится закрытой и рисуется так, как если бы для этого применялось толстое перо. Закрытая треугольная фигура преобразуется в два контура. Один из полученных контуров прошел бы по внутреннему, а другой — по внешнему контуру толстой линии, если бы треугольник был нарисован толстым пером. Конечно, все эти внутренние завитушки и уголки выглядят странно, но таковы особенности работы алгоритма метода *Widen*.

Две картинки в центре почти не отличаются от левых за исключением того, что они нарисованы более толстым пером.

Область в центре залитой фигуры, расположенной в правом верхнем углу, осталась свободной в результате применения режима заливки по умолчанию — *FillModeAlternating*. Если изменить режим заливки на *FillModeWinding*, все внутренние области фигуры будут залиты. Самая интересная версия этой фигуры — в нижнем правом углу. Она нарисована в результате применения метода *FillPath* на утолщенном контуре. Очень похожая фигура получится, если при помощи метода *DrawPath* прорисовать исходный контур толстым пером.

Метод ***GetBounds*** позволяет определить наименьший прямоугольник, в котором может поместиться контур, с учетом или без учета матричного преобразования и применения толстого пера:

Методы *GetBounds* класса *GraphicsPath*

```
RectangleF GetBounds()
RectangleF GetBounds(Matrix matrix)
RectangleF GetBounds(Matrix matrix, Pen pen)
```

Аргументы этих методов не влияют на координаты, хранимые в контуре. Имейте в виду: размеры вычисляемого прямоугольника отражают минимальные и максимальные значения координат *x* и *y* для любой точки контура. Если контур содержит кривые Безье, то прямоугольник отражает координаты управляющих точек, а не реальной кривой. Чтобы определить размеры фигуры более точно, перед *GetBounds* надо вызвать метод *Flatten*.

В главе 7 сказано, что матричное преобразование — *это линейное* преобразование. Свойство линейности налагает ряд ограничений на действия, которые можно выполнить при помощи преобразования. Например, параллелограммы могут быть преобразованы только в параллелограммы.

Сейчас мы познакомимся еще с одним преобразованием в лице метода *Warp* из класса *GraphicsPath*. Подобно *Transform* метод *Warp* модифицирует все координаты контура. Но преобразование, выполняемое *Warp*, нелинейно, и это единственное нелинейное преобразование в GDI+.

Для использования *Warp* нужно задать 4 исходные и 4 целевые координаты. Метод сопоставляет 4 исходных координаты соответствующим целевым координатам. Исходные координаты задают в виде структур *RectangleF*. Удобно (но не обязательно) заносить в аргумент *RectangleF* структуру *RectangleF*, возвращаемую *GetBounds*. Целевые координаты задают в виде массива структур *PointF*:

Методы *Warp* класса *GraphicsPath*

```
void Warp(PointF[] aptfDst, RectangleF rectfSrc)
void Warp(PointF[] aptfDst, RectangleF rectfSrc, Matrix matrix)
void Warp(PointF[] aptfDst, RectangleF rectfSrc, Matrix matrix, WarpMode wm)
void Warp(PointF[] aptfDst, RectangleF rectfSrc, Matrix matrix,
WarpMode wm, float fFlatness)
```

Эти методы также позволяют задать объект *Matrix* и коэффициент сглаживания. Исходные точки преобразуются в целевые так:

- *aptfDst[0]*— целевая точка, задающая положение левого верхнего угла прямоугольника:
- *aptfDst[1]*— целевая точка, задающая положение правого верхнего угла прямоугольника;
- *aptfDst[2]*— целевая точка, задающая положение левого нижнего угла прямоугольника:
- *aptfDst[3]*— целевая точка, задающая положение правого нижнего угла прямоугольника.

Необязательный аргумент задаст способ расчета промежуточных точек:

Перечисление *WarpMode*

Член	Значение
Perspective	0
Bilinear	1

Программа PathWarping позволяет поэкспериментировать с функцией *Warp*. Конструктор формы создает контур в виде шахматной доски размером 8X8 клеток. Затем мышью можно указать целевые точки для углов этого контура,

PathWarping.cs

```
//---------------------------------------
// PathWarping.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class PathWarping: Form
{
    ; MenuItem    miWarpMode;
     GraphicsPath path;
     PointF[]    aptfDest = new PointF[4];

     public static void Main()
     {
          Application.Run(new PathWarping());
     }
```

см. след. стр.

```
public PathWarping()
{
     Text = "Path Warping";

          // Создать меню.

     Menu = new MainMenu();

     Menu.MenuItems.Add("&Warp Mode");
     EventHandler ehClick = new EventHandler(MenuWarpModeOnClick);

     miWarpMode - new MenuItem("&" + (WarpMode)0, ehClick);
     miWarpMode.RadioCheck = true;
     miWarpMode.Checked = true;
     Menu.MenuItems[0].MenuItems.Add(miWarpMode);

     MenuItem mi * new MenuItem("&" + (WarpMode)1, ehClick);
     mi.RadioCheck = true;
     Menu.MenuItems[0].MenuItems.Add(mi);

          // Создать контур.

     path - new GraphicsPath();

     for (int i = 0; i <= 8; i++)
     {
          path.StartFigure();
          path.AddLine(0, 100 * i, 800, 100 * i);
          path.StartFigure();

          path.AddLine(100 * i, 0, 100 * i, 800);
     }
          // Инициализировать массив Point.

     aptfDest[0] = rcew Point( 50, 50);
     aptfDest[1] = new Point(200, 50);
     aptfDest[2] = new Point( SO, 200);
     aptfDest[33 = new Point(200, 200);
}
void MenuWarpModeOnClick(object obj, EventArgs ea)
{
     miWarpMode.Checked - false;
     miWarpMode = (MenuItem) obj;
     miWarpMode.Checked - true;

     Invalidate();
}
protected override void OnMouseDown(MouseEventArgs mea)
{
     Point pt;
```

см. след. стр.

```
        if (mea.Button == HouseButtons.Left)
        {
            if (ModifierKeys == Keys.None)
                pt = Point.Round(aptfDest[0]);
            else if (ModifierKeys == Keys.Shift)
                pt = Point.Round(aptfDest[2]);
            else
                return;
        }
        else if (mea.Button == MouseButtons.Right)
        {
            if (ModifierKeys == Keys.None)
                pt = Point.Round(aptfDest[1]);
            else if (ModifierKeys = Keys.Shift)
                pt = Point.Round(aptfDest[3]);
            else
                return;
        }
        else
            return;

        Cursor.Position = PointToScreen(pt);
    }
    protected override void OnMouseMove(MouseEventArgs mea)
    {
        Point pt = new Point(mea.X, mea.Y);

        if (mea.Button == MouseButtons.Left)
        {
                        if (ModifierKeys == Keys.None)
                aptfDest[0] = pt;
            else if (ModifierKeys == Keys.Shift)
                aptfDest[2] = pt;
            else
                return;
        }
        else if (mea.Button == MouseButtons.Right)
        {
            if (ModifierKeys == Keys.None)
                aptfDest[1] = pt;
            else if (ModifierKeys == Keys.Shift)
                aptfDest[3] = pt;
            else
                return;
        }
        else
            return;

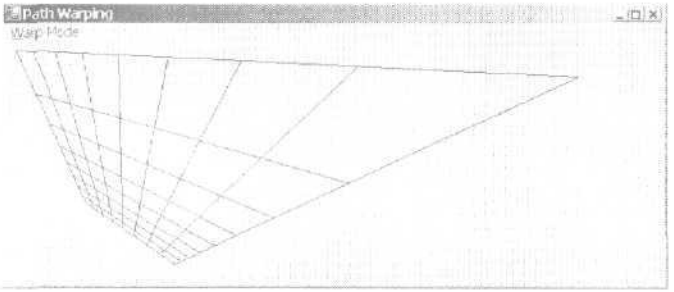
        Invalidate();
    }
```

см. след. стр.

```
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics     grfx       = pea.Graphics;
        GraphicsPath pathWarped = (GraphicsPath) path.Clone();
        WarpMode     wm         = (WarpMode) miWarpMode.Index;

        pathWarped.Warp(aptfDest, path.GetBounds(), new Matrix(), wm);
        grfx.DrawPath(new Pen(ForeColor), pathWarped);
    }
}
```

Левая и правая кнопки мыши позволяют устанавливать целевые координаты точек, в которых располагаются верхние углы прямой ольника, а эти же кнопки с нажатой клавишей Shift — координаты нижних углов прямоугольника. Меню позволяет переключаться между режимами *Perspective* и *Bilinear*. (Обратите внимание на рациональный прием: метод *OnPaint* приводит значение свойства *Index* для выбранного пользователем элемента меню к члену перечисления *WarpMode*.) Вот пример применения метода *Warp* в режиме *Perspective*:



С помощью контура удобно реализовывать собственные нелинейные преобразования. Прежде всего следует сохранить фигуру, которую нужно вывести, в виде контура. Далее надо получить все точки координат контура из свойств *PathPoints* и *PathTypes*. Модифицировать эти точки как угодно, после чего при помощи одного из конструкторов объекта *GraphicsPath*, отличного от заданного по умолчанию, нужно создать новый контур на основе модифицированного массива. В главе 19 я покажу пару примеров использования этой методики.

Отсечение при помощи контуров

Помимо рисования и заливки, контуры позволяют определять отсеченные области для объекта *Graphics*.

Методы *SetClip* класса *Graphics*

```
void SetClip(GraphicsPath path)
void SetClip(GraphicsPath path, CombineMode cm)
```

Пусть в контуре содержится эллипс. При вызове первой версии метода *SetClip* вывод всех элементов, которые будут нарисованы затем, будет ограничен областью, заданной этим эллипсом. О второй версии метода *SetClip* я расскажу чуть позже.

А сейчас давайте сразу перейдем к демонстрационной программе. Программа Clover определяет контур из четырех перекрывающихся эллипсов, задающий отсеченную область.

Clover.cs

```
//-------------------------------------
// Clover.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class Clover : PrintableForm
{
     public new static void Main()
     {
          Application.Run(new Clover());
     }
     public Clover()
     {
          Text = "Clover";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          GraphicsPath path = new GraphicsPath();

          path.AddEllipse(0,      cy / 3, cx / 2, cy / 3);  // Left
          path.AddEllipse(cx / 2, cy / 3, cx / 2, cy / 3);  // Right
          path.AddEllipse(cx / 3, 0,      cx / 3, cy / 2);  // Top
          path.AddEllipse(cx / 3, cy / 2, cx / 3, cy / 2);  // Bottom

          grfx.SetClip(path);
          grfx.TranslateTransform(cx / 2, cy / 2);

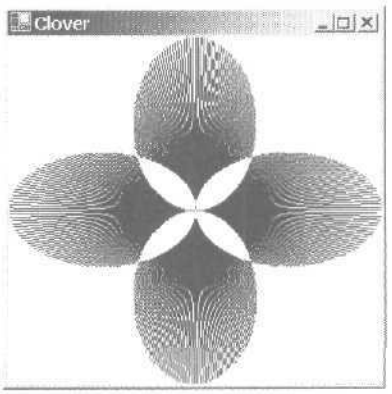
          Pen pen = new Pen(clr);
          float fRadius = (float) Math.Sqrt(Math.Pow(cx / 2, 2) +
                                            Math.Pow(cy / 2, 2));

          for (float fAngle = 0; fAngle < (float) Math.PI * 2;
                                 fAngle += (float) Math.PI / 180)
          {
               grfx.DrawLine(pen, 0, 0, fRadius * (float) Math.Cos(fAngle),
                                        -fRadius * (float) Math.Sin(fAngle));
          }
     }
}
```

Метод *DoPage* создает объект *GraphicsPath*. Созданный контур состоит из 4 эллипсов, размер которых зависит от размера клиентской области или страницы

принтера. Метод *Set-Clip* задаст для объекта *Graphics* отсеченную область, ограниченную созданным контуром,

Далее *DoPage* указывает начало координат в центре области рисования и рисует 360 линий, расходящихся от центра во все стороны. Линии обрезаются по границам внутренних областей эллипсов:



Нарисовать такую картинку любым другим способом было бы куда труднее. Заметьте: в отсеченную область не входят области перекрытия эллипсов. Это результат режима заливки контуров, заданного по умолчанию, — *FillMode**Alternate*. Если перед вызовом *SetClip* изменить режим заливки:

```
path.FillMode = FillMode.Winding;
```

области перекрытия эллипсов станут частью отсеченной области,

Алгоритм отсечения часто работает медленно. Я породил класс *Clover* от *Pi* *ntableForm*, так что можно распечатать картинку, щелкнув клиентскую область, но знайте: программа может печатать изображение час или даже дольше.

Естественно, возникает вопрос: как преобразование страницы и глобальное преобразование повлияют на отсеченную область?

При вызове *SetClip* предполагается, что координаты контура — это глобальные координаты. Последние преобразуются в координаты на устройстве так же, как при прорисовке или заливке контура. При этом границы отсеченной области сохраняются в координатах устройства и остаются таковыми. Например, после вызова *SetClip* в программе Clover можно как угодно менять преобразование страницы и глобальное преобразование, но рисование линий по-прежнему будет ограничено той же областью окна. Фактически после вызова метода *TranslateTransfrm* расположение отсеченной области в программе Clover не изменилось.

Вторая из версий *SetClip* позволяет дополнить существующую отсеченную область новой, заданной этим методом;

Перечисление *CombineMode*

Член	Значение	Описание
Replace	0	Отсеченная область = новая
Intersect	1	Отсеченная область = новая Л существующая
Union	2	Отсеченная область = новая V существующая

Перечисление *CombineMode* *(продолжение)*

Член	Значение	Описание
Xor	3	Отсеченная область = (новая + существующая) – область их перекрытия
Exclude	\	Отсеченная область = существующая -- новая
Complement	5	Отсеченная область = новая - существующая

Следующая программа создает отсеченную область на основе двух перекрывающихся эллипсов. Команда мелю позволяет выбирать значение перечисления *CombineMode*, определяющее способ объединения двух эллипсов. После этого программа закрашивает клиентскую область целиком. Как и в PathWarping, я присвоил каждому пункту меню значение от 1 до 5, соответствующее одному из значений перечисления *CombineMode*.

ClippingCombinations.cs

```
//-----------------------------------------------
// ClippingCombinations.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ClippingCombinations: PrintableForm
{
     string   strCaption = "CombineMode = ";
     MenuItem miCombineMode;

     public new static void Main()
     {
          Application.Run(new ClippingCombinations());
     }
     public ClippingCombinations()
     {
          Text = strCaption + (CombineMode)0;

          Menu = new MainMenu();
          Menu.MenuItems.Add("&CombineMode");

          EventHandler ehClick = new EventHandler(MenuCombineModeOnClick);

          for (int i = 0; i < 6; i++)
          {
               Henultem mi   = new MenuItem("&" + (CombineMode)i);
               mi.Click     += ehClick;
               mi.RadioCheck = true;

               Menu.MenuItems[0].MenuItems.Add(mi);
          }
```

см. след. стр.

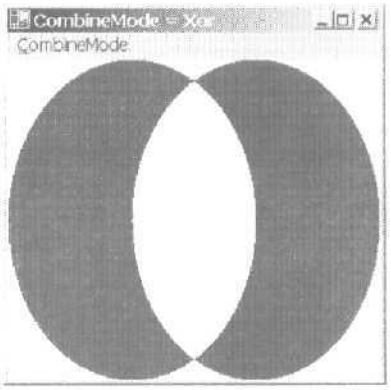
```
            miCombineMode = Menu.MenuItems[0].MenuItems[0];
            miCombineMode.Checked = true;
        }
        void MenuCombineModeOnClick(object obj, EventArgs ea)
        {
            miCombineMode.Checked = false;
            miCombineMode = (MenuItem) obj;
            foiCombineMode, Checked = true;

            Text = strCaption + (CombineMode)miCombineMode.Index;
            Invalidate();
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            GraphicsPath path = new GraphicsPath();
            path.AddEllipse(0, 0, 2 * cx / 3, cy);
            grfx.SetClip(path);

            path.Reset();
            path.AddEllipse(cx / 3, 0, 2 * cx / 3, cy);
            grfx.SetClip(path, (CombineMode)miCombineMode.Index);

            grfx.FillRectangle(Brushes.Red, 0, 0, cx, cy);
        }
}
```

Вот результат выполнения программы при выборе режима объединения *CombineModeXor*:



Дополнительные версии метода позволяют задавать (или комбинировать) отсеченные области при помощи прямоугольника:

Методы *SetClip* класса *Graphics* (выборочно)

```
void SetClipCRectangle rect)
void SetClip(Rectangle rect, CombineMode cm)
void SetClip(RectangleF rectf)
void SetClip(RectangleF rectf, CombineMode cm)
```

В классе *Graphics* также есть методы *IntersectClip* и *ExcludeClip*, которые позволяют модифицировать существующую отсеченную область. Чтобы задать обычную (т. е. бесконечно большую) отсеченную область, вызовите метод;

Метод *ResetClip* класса *Graphics*

```
void ResetClip()
```

Отсечение растровых изображений

Функция отсечения позволяет вырезать из растровых изображений области, форма которых отличается от прямоугольной. Следующая программа загружает изображение и создает в своем конструкторе контур. В методе *DoPage* она задаст отсеченную область на основе контура и рисует растровое изображение,

KeyholeClip.cs

```
//---------------------------------------------
// KeyholeClip.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class KeyholeClip: PrintableForm
{
     protected Image        image;
     protected GraphicsPath path;

     public new static void Main()
     {
          Application.Run(new KeyholeClip());
     }
     public KeyholeClip()
     {
          Text = "Keyhole Clip";

          image = Image.FromFile(
               "..\\..\\..\\..\\Images and Bitmaps\\Apollo11FullColor.jpg");

          path = new GraphicsPath();
          path.AddArc(80, 0, 80, 80, 45, -270);
          path.AddLine(70, 180, 170, 180);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.SetClip(path);
          grfx.DrawImage(image, 0, 0, image.Width, image.Height);
     }
}
```

Результат выглядит несколько нелепо (откуда на Луне замочная скважина?), но все работает:



Ясно, что в этой программе я определил контур на основе размеров заданного изображения [исходя из предположения, что его размер будет выражен в пикселах, а начало координат будет в точке (0, 0), расположенной в левом верхнем углу изображения].

Но допустим, что нужно нарисовать отсеченную область в центре клиентской области программы. Нарисовать изображение в центре нетрудно, по как центрировать контур? Одно из решений — создать контур заново на основе размеров клиентской области. Другое решение — транслировать контур или вызвать методы, транслирующие отсеченную область:

Методы *TranslateClip* класса *Graphics*

```
void TranslateClip(int cx, int cy)
void TranslateClip(float cx, float cy)
```

Программа KeyholeClipCentered переопределяет KeyholeClip и располагает как отсеченную область, так и контур по центру клиентской области программы.

KeyholeClipCentered.cs

```
//-----------------------------------------------
// KeyholeClipCentered.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class KeyholeClipCentered: KeyholeClip
{
     public new static void Main()
     {
          Application.Run(new KeyholeClipCentered());
     }
     public KeyholeClipCentered()
```

см. след. стр.

```
    {
        Text += " Centered";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.SetClip(path);

        RectangleF rectf   = path.GetBounds();
        int        xOffset = (int)((cx - rectf.Width)  / 2 - rectf.X);
        int        yOffset = (int)((cy - rectf.Height) / 2 - rectf.Y);

        grfx.TranslateClip(xOffset, yOffset);
        grfx.DrawImage(image, xOffset, yOffset, image.Width, image.Height);
    }
}
```

Кроме того, можно создать новую растровую картинку на основе размеров отсеченного изображения и применить эффект прозрачности, чтобы получить тот же результат. Эту методику демонстрирует программа KeyholeBitmap.

KeyhoEeBitmap.cs

```
//---------------------------------------
// KeyholeBitmap.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.Windows.Forms;

class KeyholeBitmap: PrintableForm
{
    Bitmap bitmap;

    public new static void Main()
    {
        Application.Run(new KeyholeBitmap());
    }
    public  KeyholeBitmap()
    {
        Text = "Keyhole Bitmap";

            // Загрузить изображение.

        Image image = Image.FromFile(
            "..\\..\\..\\..\\Images and Bitmaps\\Apollo11FullColor.jpg");
            // Создать контур отсеченной области.

        GraphicsPath path = new GraphicsPath();
```

см, след. стр.

```
            path.AddArc(80, 0, 80, 80, 45, -270);
            path.AddLine(70, 180, 170, 180);

                // Получить размер контура отсеченной области.

            RectangleF rectf = path.GetBounds();

                // Создать новое изображение, инициализированное как прозрачное.

            bitmap = new Bitmap((int) rectf.Width, (int) rectf.Height,
                                PixelFormat.Format32bppArgb);

                // Создать объект Graphics на основе нового растрового
                // изображения.

            Graphics grfx = Graphics.FromImage(bitmap);

                // Нарисовать исходное изображение на новом,
                // используя отсечения.

            grfx.SetClip(path);
            grfx.TranslateClip(-rectf.X, -rectf.Y);
            grfx.DrawImage(image, (int) -rectf.X, (int) -rectf.Y,
                           image.Width, image.Height);
            grfx.Dispose();
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            grfx.DrawImage(bitmap, (cx - bitmap.Width) / 2,
                                   <cy - bitmap.Height) / 2,
                                   bitmap.Width, bitmap.Height);
        }
}
```

Загрузка изображения и создание контура выполняется в конструкторе так же, как и в KeyholeClip. Далее программа получаст размер контура и использует полученное значение для создания нового объекта *Bitmap*. Исчисление размеров растрового изображения в пикселах задано аргументом *Format32bppArgb* (но умолчанию). Все точки растрового изображения инициализируются нулевыми значениями. Это значит, что первоначально изображение будет прозрачным, а все, что будет на нем нарисовано, — пет.

Далее конструктор получает объект *Graphics* для растрового изображения и задает отсеченную область при помощи контура. Проблема, однако, в том, что новая картинка меньше, чем загруженная, поэтому контур будет ориентирован неверно. Метод *TranslateClip* правильно располагает отсеченную область, а ***Drawl mage*** (вызванный с теми же коэффициентами смещения, что и *TranslateClip*) рисует загруженное изображение поверх нового.

Метод *DoPage* просто центрирует изображение в области экрана. Программа также может сохранить новое изображение в файле.

Области и отсечение

Исторически поддержка областей появилась в Windows гораздо раньше, чем поддержка контуров. Области были доступны уже в Windows 1.0 (1985 г.), а контуры появились только в 32-разрядных версиях Windows, начиная с Windows NT 3.1 (1993 г.) и Windows 95 (1995 г.).

С появлением контуров области во многом утратили свою важность в Windows-программировании. Их можно было бы игнорировать вовсе, если б не ключевая роль, которую они играют в отсечении. В основном контур, заданный для области отсечения, преобразуется в область. Так что чем глубже вы погружаетесь в отсечение, тем больше нужно знать про области.

Как известно, графический контур — это набор прямых и кривых. Область описывает участок на устройстве вывода. Контур довольно просто можно преобразовать в область. Фактически один из конструкторов класса *Region* (определяемого в пространстве имен *System.Drawing*) создает область прямо из контура;

Конструкторы *Region* (выборочно)

```
Region(GraphicsPath path)
```

Для исполнения этого конструктора все открытые фигуры закрываются. Создаваемая область включает внутренние области всех фигур контура. Если фигуры перекрываются, режим заливки контура определяет, какие внутренние участки фигур станут частью области, а какие нет. Только один метод класса *Graphics* использует область для рисования:

Метод *FillRegion* класса *Graphics*

```
void FillRegion(Brush brush, Region rgn)
```

Если эта область создана из контура, метод дает тот же результат, что вызов метода *FillPath* на исходном контуре.

Только одна версия *SetClip* использует область непосредственно:

Методы *SetClip* класса *Graphics*

```
void SetClip(Region rgn, CombineMode cm)
```

Может показаться странным, что во всех версиях *SetClip* рядом с аргументом-областью присутствует аргумент *CombineMode*. Дело в том, что свойство *Clip* объекта *Graphics* само по себе определяется как *Region*. Вот три свойства объекта *Graphics*, связанные с отсечением:

Свойства *Graphics* (выборочно)

Тип	Свойство	Доступ
Region	*Clip*	Чтение/запись
RectangleF	*ClipBounds*	Чтение
bool	*IsClipEmpty*	Чтение

Таким образом, отсеченную область из объекта *Region* задают не вызовом:

```
grfx.SetClip(rgn);    // Нет такого метода!
```

а установкой свойства *Clip:*

```
grfx.Clip = rgn;
```

Свойство *ClipBounds* указывает наименьший прямоугольник, заключающий в себе отсеченную область, а *IsClipEmpty* показывает, определяет ли отсеченная область несуществующий участок.

С отсечением связаны еще два свойства объекта *Graphics*:

Свойства *Graphics* (выборочно)

Тип	Свойство	Доступ
RectangleF	*VisibleClipBounds*	Чтение
bool	*IsVisibleClipEmpty*	Чтение

Для нового объекта *Graphics* свойство *VisibleClipBounds* указывает размер поверхности для рисования, для формы — размер ее клиентской области, а для принтера — размер области печати страницы. Свойство *ClipBounds* указывает прямоугольник с «бесконечными» границами (на самом деле этот прямоугольник не бесконечный, а просто очень большой).

Когда вы задаете для объекта *Graphics* отсеченную область, значение *VisibleClipBounds* будет равно области перекрытия прямоугольников, заданных исходными значениями свойств *VisibleClipBounds* и *ClipBounds*. Если отсеченная область полностью вписывается в область экрана, то значения *VisibleClipBounds* и *ClipBounds* будут равны.

Если свойство *IsClipEmpty* равно *true,* то *IsVisibleClipEmpty* также будет равно *true*. Однако возможно, что *IsClipEmpty* будет равно *false,* но отсеченная область будет за пределами границ видимой области (или области печати страницы принтера). В этом случае значение *IsVisibleClipEmpty* будет равно *true,* так как ни один из участков отсеченной области не будет располагаться в пределах области экрана.



ГЛАВА

16

Диалоговыеокна

Понятно, что на главной форме приложения можно поместить кнопки и другие элементы управления так же, как на диалоговом окне. Чем же от него отличается форма? С точки зрения управления событиями диалогового окна и его дочерних элементов, отличия невелики. Диалоговые окна составляют значительный раздел концепции программирования для Windows. Однако в библиотеке Windows Forms для диалоговых окон даже нет своего класса — просто создается еще один экземпляр (или чаще всего — еще один подкласс) класса *Form*.

Диалоговые окна бывают *модальные* (modal) или *немодальные* (modeless). Наиболее распространены первые. Как следует из названия, такие окна изменяют режим[1] ввода приложения с ввода в главную форму на ввод в диалоговое окно. Когда в программе показывается модальное диалоговое окно, пользователь не может переключаться между диалоговым окном и другими формами программы. Он должен явно закрыть диалоговое окно, что обычно делается щелчком кнопки OK [иногда Open (Открыть) или Save (Сохранить)] или Cancel (Отмена). Однако пользователь может переключаться в другие программы, не закрыв диалоговое окно. Некоторые диалоговые окна — *системные модальные* (system modal) — даже не позволяют переключаться в другие программы. Они сообщают о серьезных проблемах, и пользователь должен их закрыть, чтобы продолжить работу в Windows. (С помощью библиотеки Windows Forms системные модальные диалоговые окна создать невозможно.)

Немодальные диалоговые окна во многом похожи на дополнительные формы, создаваемые программой. (Например, две формы создаются в программах TwoForms и PaintTwoForms в главе 2.) Можно переключаться между немодальными диалоговыми окнами, создаваемыми программой, и главной формой приложения.

---

[1] От англ. **mode** — режим. — *Прим. перев.*

Программисты часто используют модальные диалоговые окна, чтобы получить от пользователя информацию, которую сложно ввести с помощью меню. (Помните программу FontMenu из главы 14?) Часто в диалоговых окнах определяются поля или свойства, позволяющие программе инициализировать диалоговые окна и (в конечном счете) получать из них информацию.

Очень часто программистам, работающим с объектно-ориентированными языками, довольно сложно решить, что должен представлять из себя объект. Один из подходов к проектированию диалоговых окон заключается в том, что окна имеют *единственное* поле (или свойство), используемое приложением для получения всей информации из диалогового окна. Это означает, что диалоговое окно связывается с определенным объектом. Такой способ структурирования программ далеко не самый худший!

В этой главе я также рассмотрю *общие* (common) диалоговые окна. В этих заранее определенных диалоговых окнах пользователь выбирает файлы, шрифты или цвета. Вы увидите, что эти диалоговые окна часто возвращают единственный объект.

Ваше первое модальное диалоговое окно

Диалоговые окна отличаются от форм приложения в основном способом вызова и (что тоже важно) завершения. В программе SimpleDialog показаны оба этих действия,

SimpleDialog.cs

```
//-----------------------------------
// SimpleDialog.cs © 2001 by Charles Petzold
//-----------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleDialog: Fern
{
    string strDisplay = "";

    public static void Main()
    {
        Application.Run(new SimpleDialog());
    }
    publicSimpleDialog()
    {
        Text = "Simple Dialog";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Dialog!", new EventHandler(MenuOnClick));
    }
    void MenuOnClick(object obj, EventArgs aa)
    {
```

см, след. стр.

```
            SimpleDialogBox dlg - new SimpleDialogBoxO;

            dlg.ShowDialog();

            strDisplay = "Dialog box terminated with " +
                         dlg.OialogResult + "!";
            Invalidate();
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics grfx = pea.Graphics;
            grfx.DrawString(strDisplay, Font, new SolidBrush(ForeColor), 0, 0);
        }
}
class SimpleDialogBox: Form
{
        public SimpleDialogBox()
        {
            Text = "Simple Dialog Box";

                // Стандартное оформление диалоговых окон

            FormBorderStyle = FormBorderStyle.FixedDialog;
            ControlBox      = false;
            MaximizeBox     = false;
            MinimizeBox     = false;
            ShowInTaskbar   = false;

                // Создаем кнопку OK.

            Button btn * new Button();
            btn.Parent   = this;
            btn.Text     = "OK";
            btn.Location = new Point(50, 50);
            btn.Size     = new Size (10 * Font.Height, 2 * Font.Height);
            btn.Click   += new EventHandler(ButtonOkOnClick);

                // Создаем кнопку Cancel.

            btn = new Button();
            btn.Parent   = this;
            btn.Text      - "Cancel";
            btn.Location = new Point(50, 106);
            btn.Size     = new Size (10 * Font.Height, 2 * Font.Height);
            btn.Click   += new EventHandler(ButtonCancelOnClick);
        }
        void ButtonOkOnClick(object obj, EventArgs ea)
        {
            DialogResult = DialogResult.OK;
        }
```

см. след. стр.

```
    void ButtonCancelOnClick(object obj, EventArgs ea)
    {
        DialogResult = DialogResult.Cancel;
    }
}
```

В этой программе два класса, каждый из которых унаследован от *Form*: *SimpleDialog* — класс главного окна программы, а *SimpleDialogBox* — класс диалогового окна программы.

В методе *Main* всего лишь создается экземпляр класса *SimpleDialog*. В конструкторе этого класса создается крохотное меню, состоящее только из одного пункта «Dialog!». Щелчок этого пункта заставляет метод *MenuOnClick* вывести диалоговое окно. Для этого сначала создается экземпляр *SimpleDialogBox:*

```
SimpleDialogBox dlg = new SimpleDialogBox();
```

Хотя мои диалоговые окна будут основываться на классах с длинными именами типа «SimpleDialogBox», я, как правило, буду использовать для обозначения диалогового окна переменную *dlg* или что-нибудь в этом роде.

При создании объекта типа *SimpleDialogBox* выполняется конструктор по умолчанию, определенный в классе *SimpleDialogBox*. В конструкторе сначала присваивается значение тексту, появляющемуся в заголовке диалогового окна:

```
Text = "Simple Dialog Box";
```

Затем присваиваются значения пяти дополнительным свойствам:

```
FormBorderStyle = FormBorderStyle.FixedDialog;
ControlBox      = false;
MaximizeBox     = false;
MinimizeBox     = false;
ShowInTaskbar   = false;
```

Присваивание этих пяти свойств общепринято для диалоговых окон. Стиль рамки *FixedDialog* не позволяет изменять размер диалогового окна, значения следующих трех свойств определяют отсутствие в заголовке меню управления (его также называют системным меню) и кнопок свертывания и развертывания. Кроме того, присваивание свойству *ControlBox* значения *false* убирает из заголовка кнопку закрытия окна. Заголовок содержит только текст диалогового окна (в данном случае «Simple Dialog Box»). Хотя у некоторых диалоговых окон Windows вообще нет заголовка, лучше, чтобы он был, так как тогда пользователь сможет переместить диалоговое окно в другое место экрана. И, наконец, присваивание значения пятому свойству выполняется, потому что не нужно показывать диалоговое окно в панели задач Windows. В панели задач следует показывать только приложения.

Далее в конструкторе создаются две кнопки со значениями свойства *Text* «OK» и «Cancel». Каждая из кнопок связана со своим обработчиком события *Click* кнопки.

Но диалоговое окно пока что не стало видимым! По завершении конструктора *SimpleDialogBox* в коде метода *MenuOnClick* класса *SimpleDialog* вызывается метод диалогового окна *ShowDialog*:

```
dlg.ShowDialog();
```

ShowDialog делает диалоговое окно видимым. Метод *ShowDialog* не возвращает управление, пока диалоговое окно не закроется.

ShowDialog — это метод, необходимый для показа модального диалогового окна. При показе модального диалогового окна нельзя переключиться обратно в главную форму программы. В этом-то и модальность! (Однако, как я уже говорил, можно переключиться в другие приложения, работающие в среде Windows.) Пока показывается модальное диалоговое окно, форма приложения не принимает ввода с клавиатуры или мыши. Однако форма продолжает принимать события *Tick* объектов *Timer* и вызовы *OnPaint*.

У двух кнопок диалогового окна имеются обработчики события *Click*: *ButtonOkOnClick* и *ButtonCancelOnClick*. Оба метода содержат только одну строку кода, присваивающую значение свойству *DialogResult* формы диалогового окна:

Свойства Form (выборочно)

Тип	Свойство	Доступ
DialogResult	*DialogResult*	Чтение/запись

Свойство *DialogResult* должно принимать значение, входящее в перечисление:

Перечисление *DialogResult*

Член	Значение
None	0
OK	1
Cancel	2
Abort	3
Retry	4
Ignore	5
Yes	6
No	7

Как видите, члены перечисления соответствуют строкам, обычно показываемым на кнопках диалоговых окон. Если эта таблица кажется знакомой, то это из-за того, что вы уже сталкивались с ней в главе 2. Метод *Show* класса *MessageBox* возвращает значение, являющееся членом перечисления *DialogResult*.

В нашей программе кнопка «OK» присваивает *DialogResult* значение *DialogResult.OK*, а «Cancel» — *DialogResult.Cancel*.

При этом в обоих случаях происходит драматическое событие: диалоговое окно закрывается и исчезает. Метод *ShowDialog*, вызвавший показ диалогового окна, возвращает управление методу *MenuOnClick*,

Хотя диалоговое окно закрыто и теперь невидимо, объект диалогового окна *dlg* по-прежнему доступен. Это значит, что *MenuOnClick* может обратиться к свойству *DialogResult* диалогового окна, чтобы определить, как закрыли диалоговое окно. В данном случае в *MenuOnClick* просто присваивается значение полю *strDisplay*, затем форма делается недействительной. Эта строка *strDisplay* показывается в методе *OnPaint*.

Теперь рассмотрим пару способов сокращения записи. Во-первых, метод *ShowDialog* определен так:

Методы ***Form*** **(выборочно)**

```
DialogResult ShowDialog()
```

Возвращаемым является значение свойства *DialogResult* в момент закрытия диалогового окна. Поэтому, когда *ShowDialog* возвращает управление, программа, показывающая диалоговое окно, может запомнить значение *DialogResult.*

```
DialogResult dr = dlg.ShowDialog();
```

Можно также поместить вызов *ShowDialog* прямо в оператор *if.*

```
if (dlg.ShowDialog() == DialogResult.OK)
{
    :
}
else
{
    :
}
```

или в оператор *switch:*

```
switch(dlg.ShowDialogO)
{
case DialogResult.OK:
    :
case DialogResult.Cancel:
    :
default:
    :
}
```

Обычно программа получает информацию из диалогового окна, если *DialogResult* равен *OK.,* и просто продолжает выполнение, если *DialogResult* равен *Cancel.*

Закрытие модального диалогового окна

Наверное, вас вполне устраивает код в программе SimpleDialogBox. Разве можно еще проще закрыть диалоговое окно, чем присваиванием значения свойству *DialogResult* в обработчике события *Click* кнопки? Читайте дальше!

Как вам, вероятно, известно из опыта, модальные диалоговые окна почти всегда закрываются, когда пользователь нажимает кнопку. Поэтому класс *Button* — или, если точнее, интерфейс *IButtonControl,* реализуемый *Button* и *LinkLabel,* — также содержит свойство *DialogResult.*

Свойство интерфейса *IButtonControl*

Тип	Свойство	Доступ
DialogResult	*DialogResult*	Чтение/запись

Мы уже знаем, что у *Form* имеется свойство *DialogResult,* а теперь вы можете видеть, что и у *Button* есть то же свойство. Обычно, если в классах *Form* и *Button* определено одно и то же свойство, есть основания предположить, что оба класса

унаследовали свойство от *Control*. Но в случае *DialogResult* это не так, хотя реализации **этих** двух свойств взаимосвязаны.

Если свойству ***DialogResult*** кнопки присваивается значение, кнопке дается инструкция: при щелчке присвоить то же значение свойству *DialogResult* предка (т. е. самого диалогового окна). Благодаря этому обработчики событий для кнопок диалогового окна не нужны, что демонстрирует программа SimplerDialog.

SimplerDialog.cs

```
//-----------------------------------------------------
// SimplerDialog.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SimplerDialog : Form
{
     string strDisplay = "";

     public static void Main()
     {
          Application.Run(new SimplerDialog());
     }
     public SimplerDialog()
     {
          Text = "Simpler Dialog";
          Menu = new MainMenu();
          Menu.MenuItems.Add("&Dialog!", new EventHandler(MenuOnClick));
     }
     void MenuOnClick(object obj, EventArgs ea)
     {
          SimplerDialogBox dlg = new SimplerDialogBox();
          DialogResult     dr  = dlg.ShowDialog();

          strDisplay = "Dialog box terminated with " + dr + "!";
          Invalidate();
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx = pea.Graphics;
          grfx.DrawString(strDisplay, Font, new SolidBrush(ForeColor), 0, 0);
     }
}
class SimplerDialogBox : Form
{
     public SimplerDialogBox()
     {
          Text = "Simpler Dialog Box";
```

см. след. стр.

```
            //Стандартное оформление диалоговых окон

        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox      = false;
        MaximizeBox     = false;
        MinimizeBox     = false;
        ShowInTaskbar   = false;

            // Создаем кнопку OK.

        Button btn = new Button();
        btn.Parent   = this;
        btn.Text     = "OK";
        btn.Location = new Point(50, 50);
        btn.Size     = new Size (10 * Font.Height, 2 * Font.Height);
        btn.DialogResult = DialogResult.OK;

            // Создаем кнопку Cancel.

        btn = new ButtonO;
        btn.Parent   = this;
        btn.Text     = "Cancel";
        btn.Location = new Point(50, 100);
        btn.Size     = new Size (10 * Font.Height, 2 * Font.Height);
        btn.DialogResult = DialogResult.Cancel;
    }
}
```

При нажатии кнопки OK или Cancel эта версия программы ведет себя так же, как SimpleDialog. Диалоговому окну не требуется явно присваивать значение свойству *DialogResult* формы, если только вы не собираетесь закрывать диалоговое о сна иными способами, не используя кнопки.

Если по-прежнему нужно выполнять обработку нажатия пользователем кнопок OK или Cancel, всегда можно установить и обработчики события *Click*. Но для закрытия диалогового окна это не требуется.

Accept и Cancel

У диалоговых окон, которые мы пока что создавали, отсутствует одна деталь, присущая типичному интерфейсу клавиатуры диалоговых окон. Вот что работает прекрасно: клавиши-стрелки или Tab позволяют перемещать фокус ввода между кнопками OK и Cancel. При перемещении фокуса кнопка, получившая фокус Е.ВОда, становится также и кнопкой по умолчанию. Можно вызвать срабатывание кнопки с фокусом ввода, нажав пробел, или срабатывание кнопки по умолчанию, нажав Enter.

Чего нельзя сделать на данный момент, так это закрыть диалоговое окно клавишей Esc. Нажатие Esc считается эквивалентным нажатию клавиши Cancel.

Более того, если на диалоговое окно поместить еще какой-нибудь элемент управления (скажем, *CheckBox*), обнаружится, что, когда *CheckBox* получает фо кус.

кнопки по умолчанию нет. Нажатие Enter при этом ни к чему не приводит. Если фокус ввода находится у элемента, не являющегося кнопкой, кнопка OK (или ее эквивалент — Open, Save и т. п.) считается кнопкой по умолчанию, т. е. именно она должна реагировать на нажатие клавиши Enter.

За этот аспект интерфейса пользователя отвечают два свойства *Form*:

Свойства *Form* (выборочно)

Тип	Свойство	Доступ
IButtonControl	*AcceptButton*	Чтение/запись
IButtonControl	*CancelButton*	Чтение/запись

Любой объект, реализующий интерфейс *IButtonControl,* в частности *Button* или *LinkLabel*, можно присвоить этим свойствам.

Свойство *AcceptButton* указывает, какой элемент управления *Button* активизируется, когда фокус находится у элемента, не являющегося кнопкой, и нажата клавиша Enter. Независимо от значения *AcceptButton* любой элемент управления, реализующий интерфейс *IButtonControl,* становится кнопкой по умолчанию и реагирует на клавишу Enter, если у него находится фокус,

Свойство *CancelButton* указывает элемент управления *Button,* который активизируется по нажатию клавиши Esc,

Возможно, *IButtonControl* вызвал ваше любопытство. Вы уже видели третью часть того, в чем заключается реализация интерфейса *IButtonControl,* — свойство *DialogResult.* Остальные две трети *IButtonControl* составляют два метода:

Методы *IButtonControl*

```
void NotifyDefault(bool bDefault)
void PerformClick()
```

Когда для элемента управления, реализующего *IButtonControl,* вызывается метод *NotifyDefault* с параметром *true,* элемент управления должен визуально показать, что является элементом по умолчанию (а значит, реагирует на нажатие клавиши Enter). Кнопка показывает, что является элементом управления по умолчанию, при помощи жирного обрамления. Метод *PerformClick* имитирует нажатие на кнопку. Это метод элемента управления *Button* по умолчанию, который форма вызывает по нажатию клавиши Enter,

Обычно свойство *DialogResult* элемента управления *Button* и свойство *AcceptButton* или *CancelButton* формы диалогового окна присваиваются в программе вместе. Так, когда на форме создается кнопка OK, Load или Save, для свойства *DialogResult* выполняется присваивание:

```
btn.DialogResult = DialogResult.OK;
```

Кроме того, свойству *AcceptButton* формы диалогового окна присваивается объект ***Button:***

```
AcceptButton = btn;
```

Аналогично, когда на форме создается кнопка Cancel, для свойства *DialogResult* кнопки выполняется присваивание:

```
btn.DialogResult = DialogResult.Cancel;
```

Л для свойства *CancelButton* выполняется присваивание:

```
CancelButton = btn;
```

Скоро я покажу программу, в которой присваиваются значения этим свойствам.

Размещение на экране

Вы. наверно, заметили, что при каждом новом запуске Windows-приложения ч;|-сто появляются в разных местах экрана. В начале сеанса работы с Windows система помещает первое приложение в левом верхнем углу экрана, Каждое следующее приложение затем размещается правее и ниже предыдущего в соответствии с каскадным шаблоном. При этом смещение равно сумме *SystemInformation.CaptionButtonSize* и *SystemInformationFrameBorderSize.*

Такое поведение прекрасно подходит для приложений. Но те же правила применяются и к диалоговым окнам с результатами, далекими от оптимальных. Получается, что диалоговое окно может появиться на некотором расстоянии от вызвавшего его приложения. Эта проблема может быть не так заметна в программах SimpleDialog и SimplerDialog, так как вы, вероятно, запускаете программу, *о:* -крываете диалоговое окно, закрываете диалоговое окно и затем — программу. При таком порядке событий диалоговое окно часто показывается с приемлемым смещением относительно формы приложения. Но если вы запустите еще несколько программ перед вызовом диалогового окна, результаты будут совсем другие.

Можно переопределить поведение по умолчанию, управляя расположением форм, используя свойство *StartPosition:*

Свойства *Form* (выборочно)

Тип	Свойство	Доступ
FormStartPosition	*StartPosition*	Чтение/запись

FormStartPosition является перечислением:

Перечисление *FormStartPosition*

Член	Значение
Manual	0
CenterScreen	1
WindowsDefaultLocation	2
WindowsDefaultBounds	3
CenterParent	4

По умолчанию для приложений Windows Forms применяется *WindowsDefaultLocation,* что означает, что Windows размещает форму в соответствии с приведенным мной описанием, но размер формы определяет само приложение. На самом деле размер формы устанавливает конструктор *Form,* но конструктор любого унаследованного от *Form* класса может переопределить размер. Поэтому все приложения Windows Forms имеют один и тот же размер по умолчанию, и он отличается от размера по умолчанию обычных программ Windows (не использующих Windows Forms).

Можно получить положение *и* размер, используемые Windows по умолчанию. указав значение *WindowsDefauUBounds.* Тогда Windows установит и положение. и размер формы. В случае *WindowsDefauUBounds* любая попытка присвоить значение положению или размеру формы в конструкторе программы будет игнорироваться, а в случае *WindowsDefaultLocation* в конструкторе можно будет присвоить значения размеру, но не положению.

CenterParent позволяет программе поместить диалоговое окно в центре формы программы без вычислений. Но размещение может оказаться неоптимальным. В этой главе я не изменял размер клиентской области приложения или диалоговое окно так, чтобы *CenterParent* приводило к тому, что диалоговое окно перекрывает (и полностью скрывает) вызвавшее его приложение. *CenterScreen* размещает диалоговое окно в центре экрана, что полезно для немодальных диалоговых окон, иногда появляющихся на экране при загрузке приложения. Как *CenterParent,* так и *CenterScreen* позволяют диалоговому окну изменять свой размер в конструкторе.

Значение *Manual* предоставляет диалоговому окну полную свободу в выборе положения и размера. Обычно это значение применяют, когда нужно расположить диалоговое окно как-либо относительно вызвавшей его формы. Лучший способ получить в диалоговом окне форму, которая его вызвала, — свойство *ActiveForm.*

В программе BetterDialog диалоговое окно располагается со смещением относительно формы приложения. Кроме того, программа демонстрирует применение свойств *AcceptButton* и *CancelButton,* о которых я говорил ранее.

BetterDialog.cs

```
//-----------------------------------------
// BetterDialQg.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BetterDialog: Form
{
     string strDisplay - "";

     public static void Main()
     {
          Application.Run(new BetterDialog());
     }
     public BetterDialog()
     {
          Text - "Better Dialog";

          Menu = new MainMenu();
          Menu.MenuItems.Add("&Dialog!", new EventHandler(MenuOnClick));
     }
```

см. след. стр.

```
    void MenuOnClick(object obj, EventArgs ea)
    {
        BetterDialogBox dlg = new BetterDialogBox();
        DialogResult    dr  = dlg.ShowDialog();

        strDisplay = "Dialog box terminated with " + dr + "!";
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        grfx.DrawString(strDisplay, Font, new SolidBrush(ForeColor), 0, 0);
    }
}
class BetterDialogBox: Form
{
    public fletterDialogBoxC)
    {
        Text = "Better Dialog Box";

            // Стандартное оформление диалоговых окон

        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox      = false;
        MaximizeBox     = false;
        MinimizeBox     = false;
        ShowInTaskbar   = false;
        StartPosition   = FormStartPosition.Manual;
        Location        = ActiveForm.Location +
                          SystemInformation.CaptionButtonSize +
                          SystemInformation.FrameBorderSize;

            // Создаем кнопку OK.

        Button btn = new Button();
        btn.Parent   = this;
        btn.Text     = "OK";
        btn.Location = new Point(50, 50);
        btn.Size     = new Size (10 * Font.Height, 2 * Font.Height);
        btn.DialogResult = DialogResult.OK;

        AcceptButton = btn;

            // Создаем кнопку Cancel,

        btn = new Button();
        btn.Parent   - this;
        btn.Text     ="Cancel";
        btn.Location = new Point(50, 100);
```

см. след. стр.

```
            btn.Size     = new Size (10 * Font.Height, 2 * Font.Height);
            btn.DialogResult = DialogResult.Cancel;

            CancelButton = btn;
        }
}
```

При выполнении стандартных операций по инициализации диалогового окна в начале конструктора свойству *StartPosition* присваивается значение *FormStartPositionManual*. Кроме того, свойству *Location* диалогового окна присваивается значение свойства *Location* активной формы (формы, вызвавшей диалоговое окно), сложенное с двумя свойствами *SystemInformation*, о которых я говорил ранее.

Кроме того, при создании в диалоговом окне каждой из двух кнопок присваиваются значения свойствам *AcceptButton* и *CancelButton* формы диалогового окна. У диалогового окна появился полноценный и корректно работающий интерфейс клавиатуры.

Окно About

Одним из общих диалоговых окон, присутствующих почти в каждом нетривиальном приложении, является *About* («О программе»). Оно может быть простым и содержать уведомление об авторском праве, а может быть сложным и показывать сведения о системных ресурсах, номер телефона и адрес Web-узла технической поддержки.

Вот программа, показывающая окно «О программе», содержащее значок, две надписи и одну кнопку.

AboutBox.cs

```
//---------------------------------------
// AboutBox.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class AboutBox: Form
{
    public static void Main()
    {
        Application.Run(new AboutBox());
    }
    public AboutBox()
    {
        Text = "About Box";
        Icon = new Icon(GetType(), "AboutBox.AforAbout.ico");

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Help");
```

см. след. стр.

```
            Menu.MenuItems[0].MenuItems.Add("&AboutAboutBox...",
                                            new EventHandler(MenuAboutOnClick));
        }
        void MenuAboutOnClick(object obj, EventArgs ea)
        {
            AboutDialogBox dlg = new AboutDialogBox();
            dlg.ShowDialog();
        }
}
class AboutDialogBox: Form
{
        public AboutDialogBox()
        {
            Text = "About AboutBox";

            StartPosition   = FormStartPosition.CenterParent;
            FormBorderStyle = FormBorderStyle.FixedDialog;
            ControlBox      = false;
            MaximizeBox     = false;
            MinimizeBox     = false;
            ShowInTaskbar   = false;

            Label label1    - new Label();
            label1.Parent   = this;
            label1.Text     = " AboutBox Version 1.0 ";
            label1.Font     = new Font(FontFamily.GenericSerif, 24,
                                       FontStyle.Italic);
            label1.AutoSize  = true;
            labell.TextAlign = ContentAlignment.MiddleCenter;

            Icon icon * new Icon(GetType(), "AboutBox.AforAbout.ico");

            PictureBox picbox = new PictureBox();
            picbox.Parent    - this;
            picbox.Image     * icon.ToBitmap();
            picbox.SizeMode  = PictureBoxSizeMode.AutoSize;
            picbox.Location  = new Point(label1.Font.Height / 2,
                                         label1.Font.Height / 2);

            Labell.Location = new Point(picbox.flight,labell.Font.Height / 2);

            int iClientWidth = label1.Right;

            Label label2    = new Label();
            label2.Parent   = this;
            label2.Text     = "\x00A9 2001 by Charles Petzold";
            label2.Font     = new Font(FontFamily.GenericSerif, 16);
            label2.Location = new Point(0, label1.Bottom +
                                        label2.Font.Height);
            label2.Size     = new Size(iClientWidth, label2.Font.Height);
```

см. след. стр.

```
        label2.TextAlign = ContentAlignment.MiddleCenter;

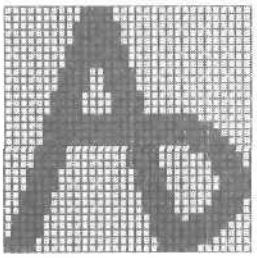
        Button button  = new Button();
        button.Parent   = this;
        button.Text     = "OK";
        button.Size     = new Size(4 * button.Font.Height,
                                   2 * button.Font.Height);
        button.Location = new Point((iClientWidth - button.Size.Width) / 2,
                                    label2.Bottom + 2 * button.Font.Height);

        button.DialogResult = DialogResult.OK;

        CancelButton = button;
        AcceptButton = button;

        ClientSize = new Size(iClientWidth,
                              button.Bottom + 2 * button.Font.Height);
    }
}
```

AforAbout.ico

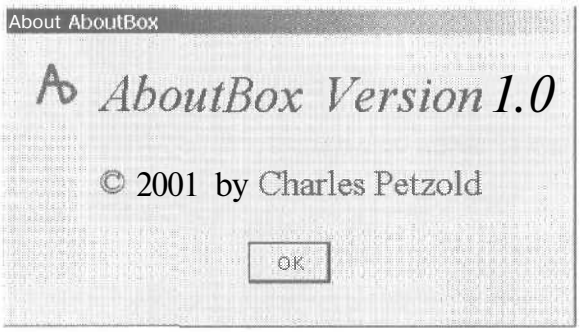


Как в конструкторе класса *AboutBox*, так и в конструкторе класса *AboutDialogBox* загружается ресурс — значок. В классе *AboutBox* ресурс делается значком формы путем присваивания значения свойству *Icon*. В классе *AboutDialogBox* для показа значка в диалоговом окне создается элемент управления *PictureBox*.

Конструктор класса *AboutDialogBox* немного усложняется позиционированием элементов управления. У первого элемента управления *Label*, содержащего имя программы, показываемое курсивом 24 пт, свойству *AutoSize* присваивается значение *true*. У элемента управления *PictureBox* свойству *SizeMode* присваивается значение *PictureBoxSizeModeAutoSize*. Графическое окно размещается в соответствии с получившимся в результате размером метки, а затем метка размещается в соответствии с получившимся в результате размером графического окна.

В диалоговых окнах с одной кнопкой ее свойству *DialogResult* обычно присваивается значение *DialogResult.OK*. Кроме того, следует присваивать эту кнопку свойствам *AcceptButton* и *CancelButton* формы.

Помимо кнопок, в таких диалоговых окнах другие элементы управления не нужны. Вместо элементов управления *Label* и *PictureBox* можно вызывать *DrawString* и *DrawIcon* в методе *OnPaint* класса *AboutDialogBox*.



На самом деле в таких простых диалоговых окнах, как это, даже не обязательно наследовать класс диалогового окна от *Form*. Вспомните главу 2, где формы создавались созданием экземпляра *Form*, а не класса-наследника *Form*. То же можно делать и с диалоговыми окнами. Совместное использование данных формой приложения и диалоговым окном становится гораздо проще, если диалоговое окно не является отдельной формой.

Проще-то проще, но программа получается уже не такой структурированной. Может, стоит писать диалоговые окна, которые можно повторно использовать в других приложениях? Это не всегда возможно, но к этому всегда следует стремиться.

Определение свойств диалоговых окон

В программе RadioButtons из главы 12 я показал, как определить группу переключателей и флажков для указания цвета эллипса и того, нужно ли обрамлять или заполнять эллипс. Давайте определим те же элементы управления, а также пару кнопок в диалоговом окне. В диалоговом окне должны быть средства, позволяющие программе инициализировать элементы управления, а также позволяющие приложению получить информацию о выборе пользователя. Обычно этот интерфейс обеспечивается открытыми свойствами диалогового окна или в некоторых случаях, в частности при недостатке времени, — открытыми полями.

В этом диалоговом окне определяются два открытых свойства, *Color* и *Fill*, предоставляющие другим классам доступ к двум параметрам, вводимым пользователем в этом диалоговом окне.

ColorFillDialogBox.cs

```
//-----------------------------------------------
// ColorFillDialogBox.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorFillDialogBox: Form
{
     protected GroupBox grpbox;
     protected CheckBox chkbox;
```

см. след. стр.

```
public ColorFillDialogBox()
{
     Text = "Color/Fill Select";

     FormBorderStyle = FormBorderStyle.FixedDialog;
     ControlBox      = false;
     MinimizeBox     = false;
     MaximizeBox     = false;
     ShowInTaskbar   = false;
     Location        = ActiveForm.Location +
                       SystemInformation.CaptionButtonSize +
                       SystemInformation.FrameBorderSize;

     string[] astrColor = { "Black", "Blue", "Green", "Cyan",
                            "Red", "Magenta", "Yellow", "White"};

     grpbox = new GroupBox();
     grpbox.Parent   = this;
     grpbox.Text     = "Color";
     grpbox.Location = new Point(8, 8);
     grpbox.Size     = new Size(96, 12 * (astrColor.Length + 1));

     for (int i = 0; i < astrColor.Length; i++)
     {
          RadioButton radiobtn = new RadioButton();
          radiobtn.Parent      = grpbox;
          radiobtn.Text        = astrColor[i];
          radiobtn.Location    = new Point(8, 12 * (i + 1));
          radiobtn.Size        = new Size(80, 10);
     }
     chkbox = new CheckBox();
     chkbox.Parent   = this;
     chkbox.Text     = "Fill Ellipse";
     chkbox.Location = new Point(8, grpbox.Bottom + 4);
     chkbox.Size     = new Size(80, 10);

     Button btn    = new Button();
     btn.Parent    = this;
     btn.Text      = "OK";
     btn.Location  = new Point(8, chkbox.Bottom + 4);
     btn.Size      = new Size(40, 16);
     btn.DialogResult = DialogResult.OK;
     AcceptButton  = btn;

     btn  = new Button();
     btn.Parent    = this;
     btn.Text      = "Cancel";
     btn.Location  = new Point(64, chkbox.Bottom + 4);
     btn.Size      = new Size(40, 16);
     btn.DialogResult = DialogResult.Cancel;
```

см. след. стр.

```
        CancelButton = btn;

        ClientSize - new Size(112, btn.Bottom + 8);
        AutoScaleBaseSize = new Size(4, 8);
    }
    public Color Color
    {
        get
        {
            for (Int i = 0; i < grpbox.Controls.Count; i++)
            {
                fladioButton radiobtn = (RadioButton) grpbox.Controls[i];

                if (radiobtn.Checked)
                    return Color.FromName(radiobtn.Text);
            }
            return Color.Black;

        }
        set
        {
            for (int i = 0; i < grpbox.Controls.Count; i++)
            {
                RadioButton radiobtn = (RadioButton) grpbox.Controls[i];

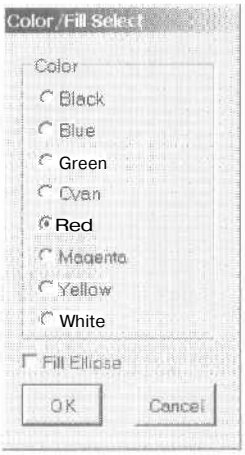
                if (value == Color.FromName(radiobtn.Text))
                {
                    radiobtn.Checked * true;
                    break;
                }
            }
        }
    }
    public bool Fill
    {
        get { return chkbox.Checked; }
        set { chkbox.Checked = value; }
    }
}
```

В конструкторе используется классическая система координат диалогового окна для создания, размещения и определения размера всех элементов управления. Достаточно гибкий код позволяет помещать в массив *astrColor* дополнительные цвета.

Объекты *GroupBox* и *CheckBox* хранятся в классе как защищенные поля, по класс не отслеживает текущее состояние переключателей и флажков. Класс использует возможность автовключения (auto-check) этих элементов управления, чтобы сохранять состояние соответствующим выбору пользователя. Но класс по-прежнгму должен предоставлять интерфейс для состояния элементов управления.

Этот интерфейс обеспечивается открытыми свойствами *Color* и *Fill*, определенными в конце программы. Свойство *Fill* довольно простое. Аксессор *get* возвращает значение свойства *Checked* флажка, аксессор *set* присваивает значение этому свойству. Однако для работы со свойством *Color* требуется поиск по всем переключателям, что равносильно поиску по всем дочерним элементам группового блока, что в свою очередь равносильно циклу по свойству *Controls*. Аксессор *get* возвращает объект *Color* выбранного в данный момент переключателя, а *set* активизирует переключатель, соответствующий данному объекту *Color*.

Вот как выглядит диалоговое окно (хотя у нас еще нет вызывающей его программы):



Конечно, оно не потребуется множеству различных приложений, но оно написано так, что допускает повторное использование. К этому всегда нужно стремиться,

В программе DrawOrFillEllipse определяется пункт меню, вызывающий это диалоговое окно. Вводимая в окно информация используется при перерисовке клиентской области, в которой показывается эллипс соответствующего цвета, возможно, залитый.

DrawOrFillEllipse.cs

```
//-----------------------------------------------
// DrawOrFillEllipse.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DrawOrFillEllipse: Form
{
     Color colorEllipse = Color.Red;
     bool  bFillElllpse = false;

     public static void Main()
```

см. след. стр.

```
    {
        Application.Run(new DrawOrFillEllipse());
    }
    public DrawOrFillEllipse()
    i
        . Text = "Draw or Fill Ellipse";
        fiesizefledraw-true;

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Options");
        Menu.MenuItems[0].MenuItems.Add("&Color...",
                                  new EventHandler(MenuColorOnClick));
    }
    void MenuColorOnClick(object obj, EventArgs ea)
    i
        ColorFillDialogBox dlg = new ColorFillDialogBox();

        dlg.Color = colorEllipse;
        dlg.Fill  = bFillEllipse;

        if (dlg.ShowDialog() == DialogResult.OK)
        {
            colorEllipse = dlg.Color;
            bFillEllipse = dlg.Fill;
            Invalidate();
        }
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics  grfx = pea.Graphics;
        Rectangle rect = new Rectangle(0, 0, ClientSize.Width - 1,
                                             ClientSize.Height - 1);
        if(bFillEllipse)
            grfx.FillEllipse(new SolidBrush(colorEllipse), rect);
        else
            grfx.DrawEllipse(new Pen(colorEllipse), rect);
    }
}
```

Обратите внимание на метод *MenuColorOnClick* — обработчик события пункта меню, вызывающего диалоговое окно. В этом методе создается объект типа *ColorFillDialogBox*. Имейте в виду, что в этот момент можно присвоить значение почти любому свойству формы диалогового окна. Так, можно изменить свойство *Text*. В диалоговых окнах, написанных для повторного использования, могут определяться другие свойства, позволяющие приложению изменять вид и функциональность диалогового окна.

Затем при инициализации диалогового окна в методе *MenuColorOnClick* двум свойствам, определенным в *ColorFillDialogBox*, присваиваются значения — поля класса *DrawOrFillEllipse*:

```
dlg.Color = colorEllipse;
dlg.Fill  = bFillEllipse;
```

Следующим оператором вызывается метод *ShowDialog* формы диалогового окна, который не вернет управление, пока диалоговое окно не закроют. При возврате управления возвращаемое *ShowDialog* значение сравнивается с *DialogResult.OK*. Если диалоговое окно закрыли щелчком кнопки OK, то в программе сохраняются новые значения свойств, и клиентская область делается недействительной:

```
if (dlg.ShowDialog() == DialogResult.OK)
{
     colorEllipse = dlg.Color;
     bFillEllipse = dlg.Fill;
     Invalidate();
}
```

Метод *MenuColorOnClick* содержит стандартный код создания, инициализации, вызова и получения информации из диалоговых окон.

Реализация кнопки Apply

В последнее время в некоторых диалоговых окнах начали появляться кнопки Apply (Применить). Кнопка Apply не закрывает диалоговое окно, но заставляет приложение применить новые значения параметров, указанные в диалоговом окне.

Кнопка Apply нарушает обычный порядок взаимодействия между приложением и модальным диалоговым окном. Требуется, чтобы приложение получало информацию о нажатии кнопки Apply, до того как метод *ShowDialog* вернет управление приложению.

Очень соблазнительно реализовать кнопку Apply, определив в приложении открытый метод, который форма диалогового окна будет вызывать при нажатии этой кнопки. Вас может удивить, почему такой путь является неправильным. А дело в том, что класс, в котором реализовано диалоговое окно, потребует от использующего его приложения реализации определенного метода с определенным именем. Знаете ли вы хоть один класс .NET, который, чтобы его задействовать, требовал бы от приложения наличия конкретного метода?

Конечно, нет. Но возможность, позволяющая классам .NET взаимодействовать с приложениями, *очень хорошо* вам знакома — это события,

Давайте перепишем класс *ColorFillDialogBox*, чтобы в нем были, во-первых, кнопка Apply, во-вторых, свойство, включающее и выключающее ее. и в-третьих, событие. Эту задачу решить проще, чем может показаться!

Ниже показан класс *ColorFillDialogBoxWithApply* — наследник *ColorFillDialogBox*. Помимо реализации кнопки Apply, в нем немного сдвигаются элементы управления и устанавливается новый размер клиентской области, чтобы уместилась новая кнопка.

ColorFillDialogBoxWithApply.cs

```
//-----------------------------------------------------
// ColorFillDialogBoxWithApply.cs © 2001 by Charles Petzold
//-----------------------------------------------------
```

см. след. стр.

```
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorFillDialogBoxWithApply: ColorFillDialogBox
{
     Button btnApply;

     public event EventHandler Apply;

     public ColorFillDialogBoxWithApply( )
     {
          flrpbox.Location = new Point(36, 8);
          chkbox.Location = new Point(36, grpbox.Bottom + 4);

          btnApply = new Button();
          btnApply.Parent   = this;
          btnApply.Enabled  = false;
          btnApply.Text     = "Apply";
          btnApply.Location = new Point(120, chkbox.Bottom + 4);
          btnApply.Size     • new Size(40, 16);
          btnApply.Click   += new EventHandler(ButtonApplyOnClick);

          ClientSize = new Size(168, btnApply.Bottom + 8);
          AutoScaleBaseSize = new Size(4, 8);
     }
     public bool ShowApply
     {
          get { return btnApply.Enabled; }
          set { btnApply.Enabled = value; }
     }
     void ButtonApplyOnClick(object obj, EventArgs ea)
     {
          if (Apply != null)
               Apply(this, new EventArgs());
     }
}
```

В начале класса вы видите оператор:

```
public event EventHandler Apply;
```

Он определят открытое событие *Apply,* основанное па делегате *EventHandler.*

Кроме того, в классе имеется повое закрытое поле *btnApply.* Очевидно, что ;>то объект *Button с текстом Apply,* создаваемый в новом конструкторе. Заметьте: в к< нструкторе присваивается значение *false* свойству *Enabled* этой кнопки. Программе, использующей это диалоговое окно, может оказаться ненужной работа с кнопкой Apply. Но как сделать ее неактивной? Конечно, с помощью открытого свойства! В классе определено новое свойство *ShowApply,* которое позволяет прогр мме, использующей диалоговое окно, активизировать/деактивизировать кнопку Apply по своему усмотрению,

Кнопка Apply не связана ни с одним значением *DialogResult*, так как она не закрывает диалоговое окно. Вместо этого я установил обработчик события *ButtonApplyOnClick* для события *Click* кнопки Apply Этот метод содержит волшебный код, необходимый для реализации события в классе. Он выполняется при нажатии пользователем кнопки Apply:

```
if (Apply != null)
    Apply(this, new EventArgs());
```

Оператор г/проверяет, имеется ли хотя бы один обработчик *события Apply,* Если да, то все установленные обработчики события вызываются с параметрами делегата *EventHandler*: первый параметр указывает источник события (форму диалогового окна), а второй — объект типа *EventArgs*.

Для большей совместимости с классами .NET следовало бы в методе *ButtonApplyOnClick* вызывать защищенный виртуальный метод *OnApply* с единственным параметром *EventArgs*, а *OnApply* содержал бы код вызова обработчиков события *Apply*.

Вот программа, использующая новую версию диалогового окна.

DrawOrFillEllipseWithApply.cs

```
//-----------------------------------------------------------
// DrawOrFillEllipseWithApply.cs © 2001 by Charles Petzold
//-----------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DrawOrFillEllipseWithApply: Form
{
     Color colorEllipse = Color.Red;
     bool  bFillEllipse = false;

     public static void Main()
     {
          Application.Run(new DrawOrFillEllipseWithApply());
     }
     public DrawOrFillEllipseWithApply()
     {
          Text = "Draw or Fill Ellipse with Apply";
          ResizeRedraw = true;

          Menu = new MainMenu();
          Menu.MenuItems.Add("&Options");
          Menu.MenuItems[0].MenuItems.Add("&Color...",
                                  new EventHandler(MenuColorOnClick));
     }
     void MenuColorOnClick(object obj, EventArgs ea)
     {
```

см. след. стр.

```
        ColorFillDialogBoxWithApply dlg =
                                new ColorFillDialogBoxWithApply();

        dlg.ShowApply = true;
        dlg.Apply    += new EventHandler(ColorFillDialogOnApply);

        dlg.Color = colorEllipse;
        dlg.Fill  = bFillEllipse;

        if (dlg.ShowDialog() == DialogResult.OK)
        {
            colorEllipse = dlg.Color;
            bFillEllipse = dlg.Fill;
            Invalidate();
        }
    }
    void ColorFillDialogOnApply(object obj, EventArgs ea)
    {
        ColorFillDialogBoxWithApply dlg *
                                (ColorFillDialogBoxWithApply) obj;

        colorEllipse = dlg.Color;
        bFillEllipse = dlg.Fill;
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics  grfx = pea.Graphics;
        Rectangle rect = new Rectangle(0, 0, ClientSize.Width - 1,
                                             ClientSize.Height - 1);
        if(bFillEllipse)
            grfx.FillEllipse(new SolidBrush(colorEllipse), rect);
        else
            grfx.DrawEllipse(new Pen(colorEllipse), rect);
    }
}
```

В методе *MenuColorOnClick* программа активизирует кнопку Apply и устанавливает для нее обработчик события:

```
dlg.ShowApply = true;
dlg.Apply    += new EventHandler(ColorFillDialogOnApply);
```

Обработчик *ColorFillDialogOnApply* приводит объект, передаваемый в качестве параметра, к типу *ColorFillDialogBoxWithApply* чтобы получить доступ к свойствам *Color* и *Fill*. Затем программа присваивает своим полям эти свойства и делает недействительной клиентскую область. Как я уже упоминал, даже когда диалоговое окно активно, приложение по-прежнему получает события *Paint*. Поэтому клиентская область перерисуется с учетом новых параметров диалогового окна.

Немодальные диалоговые окна

Пока что мы рассматривали модальные диалоговые окна, которые, конечно, используются чаще. Немодальные диалоговые окна позволяют пользователю переключаться между диалоговым окном и формой, которая создала это окно.

Немодальные диалоговые окна предпочтительней, когда пользователю удобно в течение некоторого времени не закрывать их. Пожалуй, самые распространенные немодальные диалоговые окна — это Find и Replace, показываемые программами обработки текстов. Пользователю может потребоваться оставить такое диалоговое окно активным на время, если нужно выполнить несколько действий по поиску и замене. При показе этого диалогового окна можно также редактировать документ, в котором выполняется поиск или замена.

Немодальное диалоговое окно, реализованное в показанном ниже классе, основано на программе CoIorScroll главы 12. Форма содержит три полосы прокрутки и шесть надписей, изменяющих свой размер и положение в соответствии с размером формы. Диалоговое окно остается активным в течение всего времени выполнения показывающей его программы, поэтому на нем нет кнопок и кнопки закрытия в заголовке. Открытое свойство *Color* предоставляет доступ к значениям полос прокрутки.

ColorScroliDialogBox.cs

```
//-------------------------------------------------
// ColorScrollDialogBox.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorScrollDialogBox: Form
{
     Label[]      alabelName  = new Label[3];
     Label[]      alabelValue = new Label[3];
     VScrollBar[] avscroll    = new VScrollBar[3];

     public event EventHandler Changed;

     public ColorScrollDialogBox()
     {
          Text = "Color Scroll Dialog Box";

          ControlBox    = false;
          MinimizeBox   = false;
          HaximizeBox   = false;
          ShowInTaskbar = false;

          Color[] acolor = { Color.Red, Color.Green, Color.Blue };

          for (int t - 0; i < 3; i++)
```

см. след. стр.

```
        {
            alabelName[i] = new Label ();
            alabelName[i].Parent = this;
            alabelName[i].ForeColor = acolor[i];
            alabelName[i].Text = "4" + acolor[i].ToKnownColor();
            alabelName[i].TextAlign * ContentAlignment.MiddleCenter;

            avscroll[i] = new VScrollBar();
            avscroll[i].Parent = this;
            avscroll[i].SmallChange = 1;
            avscroll[i].LargeChange = 16;
            avscroll[i].Minimum  = 0;
            avscroll[i].Maximum = 255 + avscroll[i].LargeChange - 1;
            avscroll[i].ValueChanged +=
                        new EventHandler(ScrollOnValueChanged);
            avscroll[i].TabStop = true;

            alabelValue[i] = new Label();
            alabelValue[i].Parent = this;
            alabelValue[i].TextAlign - ContentAlignment.MiddleCenter;
        }

        OnResize(EventArgs.Empty);
    }
    public Color Color
    {
        get
        {
            return Color.FromArgb(avscroll[0].Value,
                                  avscroll[1].Value,
                                  avscroll[2].Value);
        }
        set
        {
            avscroll[0].Value = value.R;
            avscroll[1].Value = value.G;
            avscroll[2].Value = value.B;
        }
    }
    protected override void OnResize(EventArgs ea)
    {
        base.OnResize(ea);

        int cx = ClientSize.Width;
        int cy = ClientSize.Height;
        int cyFont = Font.Height;

        for (int i = 0; i < 3; i++)
        {
            alabelName[i].Location - new Point(i * cx / 3, cyFont / 2);
```

см. след. стр.

```
            alabelName[i].Size = new Size(cx / 3, cyFont);

            avscroll[i].Location = new Point((4 * i + 1> * cx / 12,
                                             2 * cyFont);
            avscroll[i].Size * new Size(cx / 6, cy - 4 * cyFont);

            alabelValue[i].Location = new Point(i * cx / 3,
                                                cy - 3 * cyFont / 2);
            alabelValue[i].Size - new Size(cx / 3, cyFont);
        }
    }
    void ScrollOnValueChanged(Object obj, EventArgs «a)
    {
        for (int i - 0; i < 3; i++)
            if((VScrollBar) obj == avscroll[i])
                alabelValue[i].Text - avscroll[i].Value.ToString();

        if (Changed != null)
            Changed(this, new EventArgs());
    }
}
```

Как мы видели, в модальных диалоговых окнах обычно не определяются собственные открытые события, если в этих окнах нет кнопки Apply. Однако немодальным диалоговым окнам почти всегда приходится взаимодействовать с вызвавшим их приложением, и события — самое лучшее средство для этого.

В классе *ColorScrollDialogBox* определено событие *Changed,* которое имеет место, когда происходит событие *ValueChanged* одной из полос прокрутки. Если бы я хотел в большей степени соблюсти стиль .NET Framework, то определил бы в классе виртуальный защищенный метод *OnChanged.* Метод *OnChanged* вызывался бы в *ScrollOnValueChanged,* а затем в *OnChanged* вызывалось бы событие *Changed.*

Для показа немодального диалогового окна не используется *ShowDialog,* который не возвращает управление, пока диалоговое окно не закроется, и это не то, что нужно в данном случае. Вместо *ShowDialog* применяется *Show* — метод класса *Form, с* которым мы сталкивались пока только в главе 2,

Другой ключевой момент реализации немодальных диалоговых окон связан со свойством:

Свойства *Form* (выборочно)

Тип	Свойство	Доступ
Form	*Owner*	Чтение/запись

Свойству *Owner* немодального диалогового окна присваивается форма приложения. При этом форма становится владельцем (owner) диалогового окна. Для немодального диалогового окна нахождение во владении формы приложения означает то, что немодальное диалоговое окно всегда появляется *поверх* формы приложения. Кроме того, при свертывании формы приложения немодальное диалоговое окно исчезает с экрана.

В следующей программе создается объект типа *ColorScrollDialogBox*, его свойству *Owner* присваивается форма приложения, свойство *Color* диалогового окна инициализируется свойством *BackColor* формы приложения, устанавливается обработчик события *Changed* диалогового окна, и затем вызывается метод *Show* диалогового окна. Диалоговое окно остается активным (и позволяет изменять цвет фона формы приложения) до закрытия приложения.

ModelessColorScrott.cs

```
//-----------------------------------------------
// ModelessColorScroll.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ModelessColor Scroll: Form
{
    public static void Main()
    {
        Application.Run(new ModelessColorScroll());
    }
    public ModelessColorScroll()
    {
        Text = "Modeless Color Scroll";

        ColorScrollDialogBox dlg * new ColorScrollDialogBox();

        dlg.Owner = this;
        dlg.Color = BackColor;
        dlg.Changed += new EventHandler(ColorScrollOnChanged);
        dlg.Show();
    }
    void ColorScrollOnChanged(object obj, EventArgs ea)
    {
        . ColorScrollDialogBox dlg - (ColorScrollDialogBox) obj;

        BackColor = dlg.Color;
    }
}
```

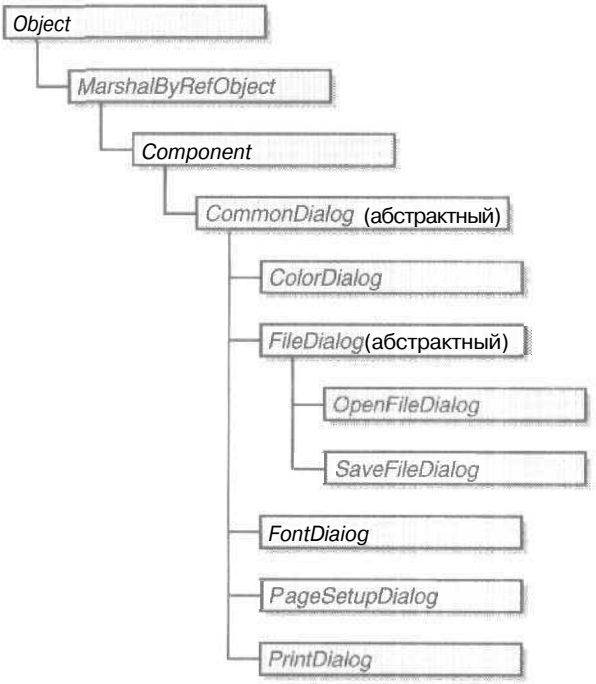
В программе Transform главы 18 демонстрируется еще одно немодальное диалоговое окно (в свою очередь вызывающее модальное диалоговое окно), позволяющее интерактивно изменять шесть элементов преобразования матрицы,

Общие диалоговые окна

Одной из основных задач Windows было внедрение стандартного пользовательского интерфейса. С первых дней Windows очень быстро установились некоторые соглашения по интерфейсу пользователя. Например, почти каждый произво-

дитель ПО начал использовать Alt+File+Open для открытия файла. Однако сами диалоги открытия файла часто значительно отличались друг от друга.

Так было, пока в Windows 3.1 не появилась *библиотека общих диалоговых окон* (common dialog box library), ставшая частью Windows API. Многое из нее доступно и в .NET Framework: диалоговые окна открытия и сохранения файлов, выбора цвета и шрифта и подготовки к печати. Иерархия классов выглядит так:



И *CommonDialog,* и *FileDialog* являются абстрактными классами, их экземпляры создать нельзя. Таким образом, остается 6 классов, которые можно использовать в приложениях. О *ColorDialog, FontDiaiog,* OpenFileDialogи *SaveFileDialog* я расскажу в этой главе, а о *PageSetupDialog* и *PrintDialog* — в главе 21, когда буду подробно рассматривать печать.

Выбор шрифта и цвета

Давайте рассмотрим и *FontDiaiog,* и *ColorDialog* в одной программе, позволяющей задавать значения свойств формы *BackColor, ForeColor* и *Font*

FontAndColorDialogs.cs

```
//-----------------------------------------------
// FontAndColorDialogs.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
```

см. след. стр.

```
using System.Windows.Forms;

class FontAndColorDialogs:Form
{
    public static void Main()
    {
        Application.Run(new FontAndColorDialogs());
    }
    public FontAndColorDialogs()
    {
        Text * "Font and Color Dialogs";
        ResizeRedraw = true;

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Format");
        Menu.MenuItems[0].MenuItems.Add("&Font...",
                                        new EventHandler(MenuFontOnClick));
        Menu.MenuItems[0].MenuItems.Add("&Background Color...",
                                        new EventHandler(MenuColorOnClick));
    }
    void MenuFontOnClick(object obj, EventArgs ea)
    {
        FontDialog fontdlg = new FontDialog();

        fontdlg.Font  = Font;
        fontdlg.Color = ForeColor;
        fontdlg.ShowColor = true;

        if(fontdlg.ShowDialog() == DialogResult.OK)
        {
            Font  = fontdlg.Font;
            ForeColor  = fontdlg.Color;
            Invalidate();
        }
    }
    void MenuColorOnClick(object obj, EventArgs ea)
    {
        ColorDialog clrdlg = new ColorDialog();

        clrdlg.Color = BackColor;

        if (clrdlg.ShowDialog() == DialogResult.OK)
            BackColor = clrdlg.Color;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics     grfx   = pea.Graphics;
        StringFormat strfmt = new StringFormat();

        strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;
```

см. след. стр.

```
            grfx.DrawString("Hello common dialog boxes!", Font,
                            new SolidBrush(ForeColor),
                            this.ClientRectangle, strfmt);
        }
    }
```

В этой программе создается меню Format, содержащее два пункта: Font и Background Color. Хотя применяются заранее созданные общие диалоговые окна, а не написанные нами, структура двух обработчиков события *Click* меню очень знакома. Рассмотрим, например, *MenuColorOnClick*. Сначала создается диалоговое окно:

```
ColorDialog clrdlg = new ColorDialog();
```

Затем оно инициализируется путем присваивания значения свойству:

```
clrdlg.Color = BackColor;
```

И затем вызывается метод *ShowDialog:*

```
if (clrdlg.ShowDialog() == DialogResult.OK)
    BackColor = clrdlg.Color;
```

Если *ShowDialog* возвращает *DialogResult.OK*, программа применяет информацию из диалогового окна.

В этой программе используются свойства формы *Font, ForeColor* и *BackColor* для инициализации двух диалоговых окон. Затем этим свойствам присваиваются значения в соответствии с тем, что укажет пользователь в диалоговом окне. Можно также определить свои поля в форме и связать с ними параметры диалогового окна.

Класс *FontDialog* позволяет пользователю выбрать шрифт и цвет шрифта, определяемые свойствами:

Свойства *FontDialog* (выборочно)

Тип	Свойство	Доступ
Font	*Font*	Чтение/запись
Color	*Color*	Чтение/запись

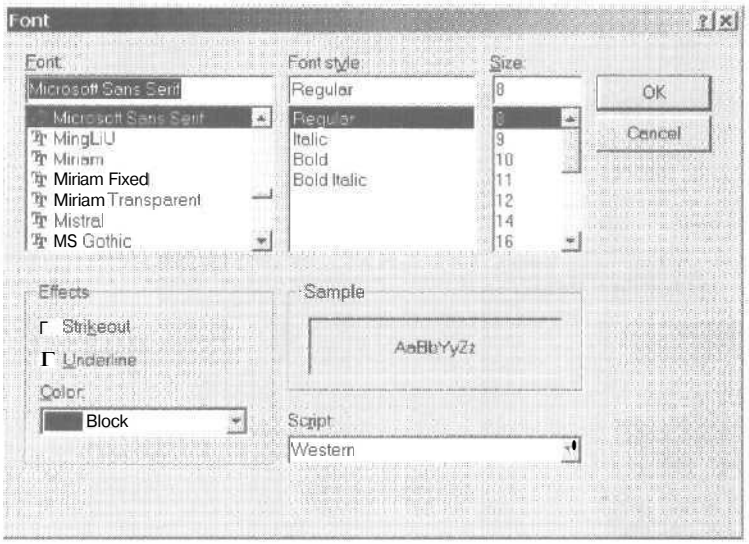
В программе FontAndColorDialogs эти два свойства просто инициализируются значениями свойств *Font* и *ForeColor* формы, а затем, если диалоговое окно закрыто кнопкой OK. этим свойствам формы присваиваются значения.

В программе FontAndColorDialogs, чтобы активизировать возможность выбора цвета, которая по умолчанию неактивна, присваивается значение свойству *ShowColor*. Следующие свойства позволяют управлять показом цвета и других элементов диалогового окна выбора шрифта:

Свойства *FontDialog* (выборочно)

Тип	Свойство	Доступ	По умолчанию
bool	*ShowEffects*	Чтение/запись	*True*
bool	*ShowColor*	Чтение/запись	*False*
bool	*ShowApply*	Чтение/запись	*False*
bool	*ShowHelp*	Чтение/запись	*False*

Если присвоить свойству *SbowEffects* значение *false*, то диалоговое окно не позволит выбрать подчеркнутый (underline) или зачеркнутый (strikeout) шрифт. Вот как выглядит диалоговое окно, когда и *SbowEffects,* и *ShowColor* равны *true*:



Свойства *ShowApply* и *ShowHelp* управляют показом кнопок Apply и Help. Если активизировать эти кнопки, может потребоваться обработка событий:

События *FontDialog* (выборочно)

Событие	Метод	Делегат	Параметр
Apply	*OnApply*	*EventHandler*	*EventArgs*
HelpRequest	*OnHelpRequest*	*EventHandler*	*EventArgs*

У вас уже есть опыт использования кнопки Apply, но я еще продемонстрирую ее применение в *FontDialog* в следующей версии программы.

Класс *FontDialog* содержит еще несколько свойств, управляющих показом шрифтов в диалоговом окне, но обычно подходят их значения по умолчанию. Однако дальше в этой главе (в программе HeadDump), я использую свойство:

Свойства *FontDialog* (выборочно)

Тип	Свойство	Доступ	По умолчанию
bool	*FixedPitchOnly*	Чтение/запись	*False*

Этому свойству присваивается значение *true,* если нужно показывать только моноширинные шрифты.

Диалоговое окно *ColorDialog* позволяет выбрать цвет, который содержится в свойстве *Color* типа *Color*.

Свойства *ColorDialog* (выборочно)

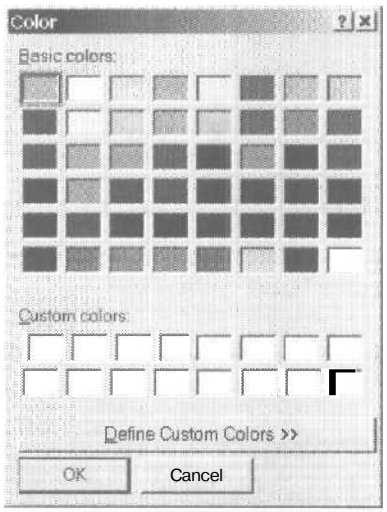
Тип	Свойство	Доступ
Color	*Color*	Чтение/запись

Несколько других свойств типа *bool* управляют различными аспектами диалогового окна:

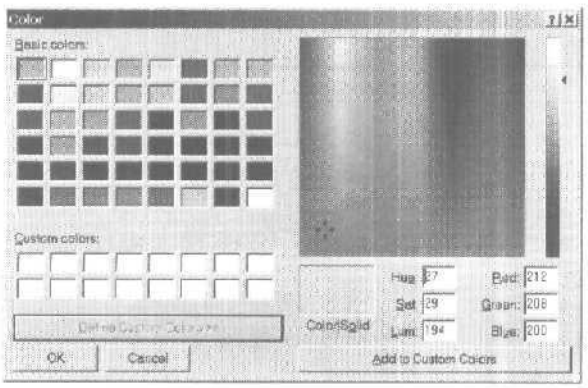
Свойства *ColorDialog* (выборочно)

Тип	Свойство	Доступ	По умолчанию
bool	*FullOpen*	Чтение/запись	*False*
bool	*AllowFullOpen*	Чтение/запись	*True*
bool	*SolidColorOnly*	Чтение/запись	*false*
bool	*AnyColor*	Чтение/запись	*false*
bool	*ShowHelp*	Чтение/запись	*false*

В классе *ColorDialog* не поддерживается кнопка Apply. Вот как по умолчанию выглядит диалоговое окно выбора цвета:



Обычно достаточно просто выбрать один из показываемых цветов и щелкнуть OK, Если ни один из этих цветов вас не устраивает, можно щелкнуть кнопку Define Custom Colors. Тогда диалоговое окно расширится:



При этом можно определить дополнительный цвет справа и добавить его в набор Custom Colors слева. Нельзя вернуть диалоговое окно в исходное состояние.

не закрыв его кнопкой OK или Cancel. Если перед вызовом *SbowDialog* присвоить свойству *FullOpen* значение *true,* диалоговое окно откроется в расширенном варианте. Если свойству *AllowFullOpen* присвоить *false,* то независимо от значения *FullOpen* диалоговое окно откроется в обычном варианте, а кнопка Define Custom Colors будет неактивна.

Для приложения, использующего класс *ColorDialog,* дополнительные цвета делает доступными свойство:

Свойства *ColorDialog* (выборочно)

Тип	Свойство	Доступ
int[]	*CustomColors*	Чтение/запись

Дополнительные цвета хранятся в массиве из шестнадцати 32-битных целых, в которых красный цвет является наименьшим значащим байтом, зеленый — следующим байтом, синий — третьим, а наибольший значащий байт равен 0. *(Не* используйте эти целые значения в статическом методе *Color.FromArgb* возвращающем объект *Color,* получаемый по целому числу. В этом методе синий является наименьшим значащим байтом, дальше идут зеленый, красный, и альфа-значение, в котором 0 означает прозрачность.)

Допустим, пользователь вызвал диалоговое окно выбора цвета из программы FontAndColorDialogs, щелкнул кнопку Define Custom Colors и тщательно подобрал 16 дополнительных цветов. Затем пользователь выбрал один из них и нажал OK. Затем он снова вызвал диалоговое окно и... Все дополнительные цвета исчезли!

Что случилось? Все очень просто. Взгляните на метод *MenuColorOnClick.* При вызове диалогового окна программа каждый раз создает его зяново. Когда метод *MenuColorOnClick* заканчивает работу, ссылок на объект *clrdlg* типа *ColorDialog* больше не остается, поэтому этот объект уничтожается при сборке мусора.

Если оставить у параметра *AllowFullOpen* значение по умолчанию *true,* то будет невежливо по отношению к пользователю не сохранять дополнительные цвета при различных вызовах диалогового окна. Это можно сделать по-разному. Можно определить массив целых значений:

```
int[]  aiCustomColors;
```

Перед вызовом диалогового окна методом *SbowDialog* можно присваивать свойству значение поля, например;

```
clrdlg.CustomColors = aiCustomColors;
```

Когда *SbowDialog* возвращает управление, то независимо от того, как пользователь закрыл диалоговое окно, дополнительные цвета запоминаются в поле:

```
aiCustomColors = clrdlg.CustomColors;
```

А можно поступить и еще проще — удалите оператор:

```
ColorDialog clrdlg = new ColorDialog();
```

из метода *MenuColorOnClick* и сделайте этот оператор определением поля. Тогда программа будет использовать только один экземпляр *ColorDialog* в течение всего выполнения приложения.

Вот усовершенствованная версия программы FontAndColorDialogs, в которой показывается кнопка Apply в диалоговом окне выбора шрифта.

BetterFontAndColorDialogs.cs

```
//-----------------------------------------------------------
// BetterFontAndColorDialogs.cs © 2001 by Charles Petzold
//-----------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BetterFontAndColorDialogs:Form
{
     protected ColorDialog clrdlg = new ColorDialog();

     public static void Main()
     {
          Application.Run(new BetterFontAndColorDialogs());
     }
     public BetterFontAndColorDialogs()
     {
          Text = "Better Font and Color Dialogs";

          Menu * new MainMenu();
          Menu.MenuItems.Add("&Format");
          Menu.MenuItems[0].MenuItems.Add("&Font...",
                                          new EventHandler(MenuFontOnClick));
          Menu.MenuItems[0].MenuItems.Add("&Background Color...",
                                          new EventHandler(MenuColorOnClick));
     1
     void MenuFontOnClick(object obj, EventArgs ea)
     {
          FontDialog fontdlg = new FontDialog();

          fontdlg.Font  = Font;
          fontdlg.Color = ForeColor;
          fontdlg.ShowColor = true;
          fontdlg.ShowApply - true;
          fontdlg.Apply += new EventHandler(FontDialogOnApply);

          if(fontdlg.ShowDialog() == DialogResult.OK)
          {
               Font = fontdlg.Font;
               ForeColor = fontdlg.Color;
               Invalidate();
          }
     }
     void MenuColorOnClick(object obj, EventArgs ea)
     {
```

см. след. стр.

```
            clrdlg.Color = BackColor;

            if (clrdlg.ShowDialog() == DialogResult.OK)
                BackColor = clrdlg.Color;
        }
        void FontDialogOnApply(object obj, EventArgs ea)
        {
            FontDialog fontdlg = (FontDialog) obj;

            Font = fontdlg.Font;
            ForeColor * fontdlg.Color;
            Invalidate();
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics grfx = pea.Graphics;
            grfx.DrawString("Hello common dialog boxes!", Font,
                            new SolidBrush(ForeColor), 0, 0);
        }
}
```

Теперь дополнительные цвета сохраняются при закрытии и повторном показе диалогового окна

К сожалению, при сохранении параметров диалогового окна от одного запуска к другому возникает вопрос: «Как сохранить изменения при закрытии и повторном запуске *программы?*».

Для этого скорее всего придется задействовать возможности реестра Windows.

Использование реестра Windows

Реестр Windows — это механизм общего назначения, который приложения (и сама ОС) используют для сохранения информации программы, которую нужно запомнить при завершении приложения. Информация хранится в иерархическом формате. Можно задействовать программу Редактор реестра (Registry Editor, Regedit.exe) из состава Windows для просмотра (и, если вы настолько отважны, даже модификации) содержимого реестра компьютера.

Информация в реестре упорядочивается при помощи *разделов* (key), которые часто записываются с применением того же синтаксиса, что и у каталогов. Так, в Registry Editor можно встретить раздел HKEY_CURRENT_ USER\Software\Microsoft\Notepad, где хранится вся информация, которую в реестр записывает Microsoft Notepad (Блокнот). У каждого из фрагментов информации имеются имя (например, iPointSize или iWindowPosX), тип (и в том, и в другом случае это REG_DWORD – 32-битное беззнаковое целое) и значение.

Работа с реестром Windows поддерживается в двух классах пространства имен *Microsoft.Win32*.Класс *Registry* состоит лишь из семи статических полей только для чтения, соответствующих семи возможным корневым разделам реестра. В столбце «Описание» показано, как эти корневые разделы определяются в файлах заголовков Win32 и показываются в Registry Editor:

Статические поля *Registry*

Тип	Свойство	Доступ	Описание
RegistryKey	*ClassesRoot*	Только чтение	HKEY_CLASSES_ROOT
RegistryKey	*CurrentUser*	Только чтение	HKEY_CURRENT_USER
RegistryKey	*LocalMachine*	Только чтение	HKEY_LOCAL_MACHINE
RegistryKey	*Users*	Только чтение	HKEY_USERS
RegistryKey	*PerformanceData*	Только чтение	HKEY_PERFORMANCE_DATA
RegistryKey	*CurrentConfig*	Только чтение	HKEY_CURRENT_CONFIG
RegistryKey	*DynData*	Только чтение	HKEY_DYN_DATA

Большинство приложений ограничивается использованием раздела *CurrentUser* для хранения информации, относящейся к конкретному пользователю, например, предпочитаемых им шрифтов, цветов и других параметров.

Второй класс называется *RegistryKey*. Вот его наиболее часто используемые методы:

Методы *RegistryKey* (выборочно)

```
RegistryKey CreateSubKey(string strSubKey)
RegistryKey OpenSubKey(string strSubKey)
RegistryKey OpenSubKey(string strSubKey, bool bWritable)
void SetValue(string strName, object obj)
object GetValue(string strName)
void Close()
```

Заметьте: методы *CreateSubKey* и *OpenSubKey* являются членами класса *RegistryKey* и возвращают объекты типа *RegistryKey*. Сначала объект *RegistryKey* берется из одного из полей класса *Registry*, например:

```
RegistryKey regkey = Registry.CurrentUser;
```

Затем можно получить другие объекты *RegistryKey*, сочетая этот раздел реестра с параметром-подразделом, передаваемым методу *CreateSubKey* или *OpenSubKey*. Например, если *regkey* получен из *Registry.CurrentUser*, то вызов:

```
regkey = regkey.OpenSubKey("Software\\Microsoft\\Notepad");
```

возвратит раздел реестра, позволяющий прочитать информацию, записанную Notepad. Можно также объединить эти два вызова в один:

```
RegistryKey regkey =
    Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Notepad");
```

Но при таком вызове возвращается раздел для Notepad. Конечно, это не тот раздел, что вы будете использовать для своего приложения. Можно создать собственный раздел с помощью метода *CreateSubKey*, например:

```
RegistryKey regkey =
    Registry.CurrentUser.CreateSubKey("Software\\MyCompany\\MyApp");
```

Скорее всего *CreateSubKey* потребуется использовать при первой установке или первом запуске программы. В дальнейшем можно задействовать *OpenSubKey*, чтобы открыть раздел для чтения:

```
RegistryKey regkey =
    Registry.CurrentUser.OpenSubKey("Software\\MyCompany\\MyApp");
```

Можно также применить *OpenSubKey,* чтобы открыть раздел для записи:

```
RegistryKey regkey =
    Registry.CurrentUser.OpenSubKey("Software\\MyCompany\\MyApp", true);
```

По окончании работы с реестром его закрывают так:

```
regkey.Close();
```

Методы *SetValue* и *GetValue* позволяют читать и записывать значения, связанные с именами. Но будьте осторожны: у вызова *SetValue* такой синтаксис, что кажется, будто в качестве второго параметра можно использовать объект любого типа, например объект типа *Font:*

```
regkey.SetValue("MyFont", font);
```

Этот вызов, казалось бы, будет работать, но когда потребуется прочитать тот же самый объект методом *GetValue,* возникнет проблема:

```
font = (Font) regkey.GetValue("MyFont");  // Не будет работать!
```

Если бы реестр изначально проектировали в расчете на поддержку объектно-ориентированного интерфейса, то эти два вызова, наверное, работали бы. Но это не так, и они не работают. В основном вам придется ограничиваться хранением строк, 32-битных целых и массивов байтов. (Массивы байтов позволяют хранить обобщенную двоичную информацию, хотя, пожалуй, они не столь удобны, как хотелось бы.)

Показанный выше вызов *SetValue* на самом деле сохранит *font.ToString()*, т. е. строку, описывающую объект *Font.* Но при вызове *GetValue* ее не удастся преобразовать в объект типа *Font,* и неправильное преобразование типов вызовет исключение времени выполнения. Чтобы сохранить объект типа *Font* в реестре, нужно сохранять все, что требуется, чтобы снова создать шрифт, в форме строк, 32-битных целых и массивов байтов.

Рассмотрим пример. Класс *DialogsWithRegistry* в показанной ниже программе является наследником *BetterFontAndColorDialogs,* и в нем добавлена поддержка реестра. Семь полей *const* определяют раздел реестра и все имена реестра, используемые в программе.

DialogsWithRegistry.cs

```
//-----------------------------------------
// DialogsWithRegistry.cs © 2001 by Charles Petzold
//-----------------------------------------
using Microsoft.Win32;
using System;
using System.Drawing;
```

см. след. стр.

```
using System.Windows.Fonrts;

class DialogsWithRegistry: BetterFontAndColorDialogs
{
     const string strRegKey =
               "Software\\ProgrammingWindowsWithCSharp\\DialogsWithRegistry";
     const string strFontFace  = "FontFace";
     eoRst string strFontSize  = "FontSize";
     const string strFontStyle = "FontStyle";
     const string strForeColor * "ForeColor";
     const string strBackColor = "BackColor";
     const string strCustomClr = "CustomColor";

     public new static void Main()
     {
          Application.Run(new DialogsWithRegistry());
     }
     public DialogsWithRegistry()
     {
          Text = "Font and Color Dialogs with Registry";

          RegistryKey regkey = Registry.CurrentUser.OpenSubKey(strRegKey);

          if (regkey != null)
          {
               Font * new Font((string) regkey.GetValue(strFontFace),
                               float.Parse(
                                    (string) regkey.GetValue(strFontSize)),
                               (FontStyle) regkey.GetValue(strFontStyle));

               ForeColor = Color.FromArgb(
                              £int) regkey.GetValue(strForeColor));

               BackColor - Color.FromArgb(
                              (int) regkey.GetValue(strBackColor));

               int[] aiColors - new int[16];

               for (int i = 0; i < 16; i++)
                    aiColors[i] = (int) regkey.GetValue(strCustomClr + i);

               clrdlg.CustomColors = aiColors;

               regkey.Close();
          }
     }
     protected override void OnClosed(EventArgs ea)
     {
          RegistryKey regkey =
                         Registry.CurrentUser.OpenSubKey(strRegKey, true);
```

см. след. стр.

```
            if (regkey == null)
                regkey = fleglstry.CyrrentUser.CreateSubKey(strRegKey);

            regkey.SetValue(strFontFace, Font.Name);
            regkey.SetValue(strFontSize, Font.SizeInPoints.ToString());
            regkey.SetValue(strFontStyle, (int) Font.Style);
            regkey.SetValue(strForeColor, ForeColor.ToArgb());
            regkey.SetValue(strBackColor, BackColor.ToArgb());

            for (int i = 0; i < 16; i++)
                regkey.SetValue(strCustomClr + i, clrdlg.CustomColors[i]);

            regkey.Close();
        }
    }
```

Сначала обсудим переопределение метода *OnClosed*, вызываемого при закрытии формы. Это удобный момент для записи информации в реестр. Если *OpenSubKey* возвращает *null*, программа запускается первый раз, поэтому вызывается *CreateSubKey* для создания раздела реестра. Каждый из вызовов *SetValue* сохраняе г в реестре или целое значение, или строку. У свойства формы *Font* нужно сохранить три значения: свойства *Name, SizeInPoints* и *Style*. Свойство *SizeInPoints* класса *font* имеет тип *float*, поэтому оно преобразуется в строковое представление методом *ToString*. Метод *ToArgb* класса *Color* используется для преобразования объектов *Color* в целые числа,

Кроме того, обратите внимание на вызов *SetValue* в цикле *for,* сохраняющем дополнительные цвета. Значением имени является:

```
strCustomClr + i
```

что создает имена CustomColor0, CustomColorl и т. д. до CustomColor15.

Значения загружаются из реестра в конструкторе программы. Шрифт формы заново создается с применением загруженных значений имени шрифта, размера в пунктах и стиля. Размер в пунктах преобразовывается из строки обратно в *float* статическим методом *Parse* структуры *Single*. Статический метод *Color.FromArgb* преобразовывает сохраненные целые значения обратно в объекты *Color*.

Так как реализация поддержки операций с реестром требует работы с двумя блоками кода, код чтения и записи в реестр может оказаться сложным для отладки. Лучше всего сначала добиться нормальной работы вызовов *SetValue*. Процесс работы с реестром позволяет наблюдать редактор реестра (клавиша F5 в нем обновляет информацию). Кроме того, он позволяет целиком удалить раздел реестра, что дает возможность проверить, как программа повторно создает записи реестра в случае их повреждения. Добившись правильной работы вызовов *SetValue,* приступайте к кодированию вызовов *GetValue*.

Диалоговое окно Open File

И *OpenFileDialog*, и *SaveFileDialog* наследуют абстрактному классу *FileDialog*, в котором определено несколько общих свойств этих двух классов. Классы *OpenFile-*

Dialog и *SaveFileDialog*являются изолированными (sealed), т. е. от них нельзя наследовать.

Оба класса предназначены в первую очередь для того, чтобы возвращать полное имя файла, указываемое пользователем путем выбора из списка или набираемое вручную. С учетом того, что приходится иметь дело с файловым вводом-выводом, применять эти диалоговые окна может оказаться сложнее, чем диалоговые окна выбора шрифта или цвета.

Рассмотрим сначала *OpenFileDialog.* Три свойства связаны с получением имени файла или нескольких имен файлов из диалогового окна;

Свойства *OpenFileDialog* (выборочно)

Тип	Свойство	Доступ
bool	*Multiselect*	Чтение/запись
siring	*FileName*	Чтение/запись
string[]	*FileNames*	Чтение

По умолчанию *Multiselect* равно *false,* т. е. пользователь может выбрать в диалоговом окне только один файл, при этом *FileName* содержит имя выбранного файла.

Обладая только этой информацией, обсудим HeadDump — программу формирования шестнадцатеричного дампа. Показывается лишь та часть дампа выбранного файла, что умещается в клиентской области. В этой программе используется статический метод *ComposeLine* программы HexDump из приложения А (в котором можно получить информацию о реализации файлов и потоков в NET).

HeadDump.cs

```
//------------------------------------------
// HeadDump.cs * 2001 by Charles Petzold
//------------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class HeadDump: Form
{
     const string strProgName  = "Head Dump";
     string       strFileName = "";

     public static void Main()
     {
          Application fltift(iuw HeadDump());
     }
     public HeadDump ()
     {
          Text * strProgName;
          Font = new Font(FontFamily.GenericMonospace,
                          Font.SizeInPoints);
```

см. след. стр.

```
        Menu = new MainMenu();
        Menu.MenuItems.Add("&File");
        Menu.MenuItems[0].MenuItems.Add("&Open...",
                              new EventHandler(MenuFileOpenOnClick));
        Menu.MenuItems.Add("F&ormat");
        Menu.MenuItems[1].MenuItems.Add("&Font...",
                              new EventHandler(MenuFormatFontOnClick));
}
void MenuFileOpenOnClick(object obj, EventArgs ea)
{
        OpenFileDialog dlg * new OpenFileDialog();

        if (dlg.ShowDialog() == DialogResult.OK)
        {
                strFileName = dlg.FileName;
                Text = strProgName + " - " + Path.GetFileName(strFileName);
                Invalidate();
        }
}
void MenuFormatFontOnClick(object obj, EventArgs ea)
{
        FontDialog dlg = new FontDialog();

        dlg.Font - FoRt;
        dlg.FixedPitchOnly = true;

        if (dlg.ShowDialog() == DialogResult.OK)
        {
                Font = dlg.Font;
                Invalidate();
        }
}
protected override void OnPaint(PaintEventArgs pea)
{
        Graphics    grfx  = pea.Graphics;
        Brush       brush - new SolidBrush(ForeColor);
        FileStream fs;

        try
        {
                fs = new FileStream(strFileName, FileMode.Open,
                                    FileAccess.Read, FileShare.Read);
        }
        catch
        {
                return;
        }

        for (int iLine = 0; iLine <= ClientSize.Height / Font.Height;
                              iLine++)
```

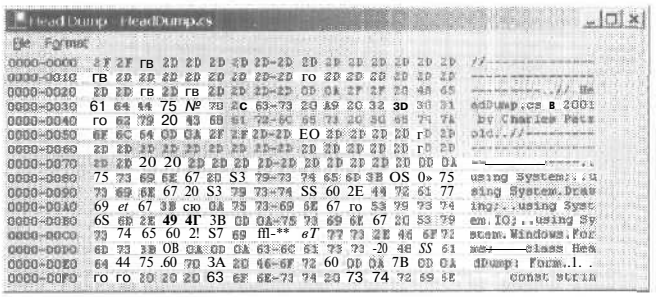
см. след. стр.

```
            {
                byte[] abyBuffer = new byte[16];
                int    iCount    = fs.Read(abyBuffer, 0, 16);
                string str       = HexDump.ComposeLine(16 * iLine,
                                                       abyBuffer, iCount);

                grfx.DrawString(str, Font, brush, 0, iLine * Font.Height);
            }
            fs.Close();
        }
    }
```

Так как при показе пропорциональным шрифтом шестнадцатеричный дамп превращается в беспорядочное нагромождение символов, свойству *Font* формы присваивается шрифт *GenericMonospace*. В меню программы можно изменить его на другой моноширинный шрифт, а также выбрать файл.

В методе *MenuFileOpenOnClick* демонстрируется, как создавать и показывать объект *OpenFileDialog*. Если *ShowDialog* возвращает *DialogResult.OK*, то программа сохраняет свойство *FileName* в поле *strFileName* и присваивает свойству *Text* формы новое значение, используя статический метод *Path.GetFileName*, чтобы извлечь имя файла из полного имени файла, содержащего путь. В методе *OnPaint* файл открывается, из файла считываются байты, выполняется их форматирование. Вот как программа показывает свой исходный файл:



При экспериментах с HeadDump вы обнаружите, что начальный каталог, показываемый в диалоге открытия файла. — каталог My Documents. Можно перейти в другие каталоги. *OpenFileDialog* автоматически сохраняет выбранный в конечном итоге каталог в реестре Windows. При следующем запуске прорраммы в диалоге открытия файла будут показываться файлы из последнего каталога, в котором вы побывали, используя это диалоговое окно.

По умолчанию при перемещении по каталогам *OpenFileDialog* изменяет текущий каталог приложения. Если нужно установить начальный каталог диалогового окна или восстанавливать текущий каталог приложения после закрытия диалогового окна, можно использовать два свойства:

Свойства *FileDialog* (выборочно)

Тип	Свойство	Доступ
string	*InitialDirectory*	Чтение/запись
bool	*RestoreDirectory*	Чтение/запись

Хотя большинство пользователей перемещается по каталогам и выбирает файлы из списка, кто-то может также ввести имя каталога или файла вручную. Следующие два свойства по умолчанию имеют значение *true*, и в этом случае диалоговое окно само проверяет допустимость пути или имени файла перед закрытием диалогового окна:

Свойства *FileDiaiog* (выборочно)

Тип	Свойство	Доступ
bool	*CheckPathExists*	Чтение/запись
bool	*CheckFileExists*	Чтение/запись

Если нужно разрешить пользователю создать новый файл при помощи *OpenFile-Dialog,* присвойте *CheckFileExists* значение *false.*

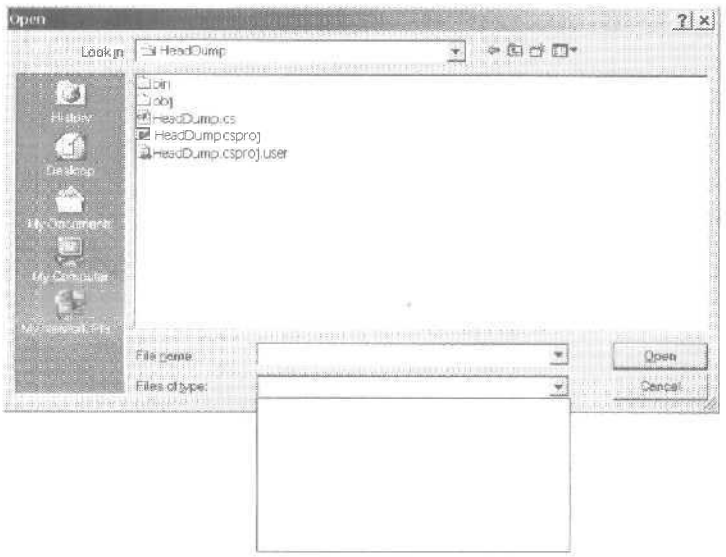
Следующие два свойства определяют, показывать ли в диалоговом окне флажок Open As Read-Only и установил ли его пользователь:

Свойства *OpenFileDialog* (выборочно)

Тип	Свойства	Доступ
bool	*ShowReadOnly*	Чтение/запись
bool	*ReadOnlyChecked*	Чтение/запись

Если этот флажок показывается и пользователь его установил, то программа не должна позволять сохранять изменения в файле на диске.

Остальные свойства, которые я хотел бы рассмотреть, связаны с типам: [и расширениями файлов. Возможно, вы заметили в программе HeadDump в *OpenFile-Dialog* пустое комбинированное окно Files Of Type:



Б HeadDump это нормально, так как программа может открывать и показывать любые типы файлов. Но в большинстве программ требуется, чтобы в *OpenFileDialog* показывались только определенные типы файлов. Это позволяет сделать строка,

называемая *фильтром* (filter). В программе HeadDump я мог бы определить фильтр, скажем, так:

```
"All Files (*.*)|*.*"
```

или так:

```
"All Files|*.*"
```

Часть строки, идущая до вертикальной черты, — это то, что диалоговое окно покажет в комбинированном окне Files Of Type. Строка справа от вертикальной черты определяет типы файлов, показываемых в комбинированном окне. В нашем примере это все файлы.

Вы вправе показывать или не показывать в части строки слева от вертикальной черты шаблон файла. Эта часть — то, что видит пользователь, она не влияет на файлы, показываемые в диалоговом окне.

Если вы пишете клон программы Notepad, что мы с вами и проделаем в главе 18, то фильтр можно определить так:

```
"Text Documents (*.txt)|*.txt|" +
"All Files|*.*"
```

В этом случае комбинированное окно Files Of Type будет содержать две строки. При выборе пользователем в комбинированном окне первой строки в диалоговом окне будут показываться только файлы с расширением *txt*; если пользователь выберет вторую строку, то в диалоговом окне будут показаны все файлы. Хотя для ясности я составил строку из двух строк, объединенных символом конкатенации (+), на самом деле это просто одна длинная строка, состоящая из частей, разделенных вертикальной чертой:

```
"Text Documents (*.txt)|*.txt|All Files|*.*"
```

Этих частей всегда вдвое больше, чем строк в комбинированном окне.

При написании программы, загружающей различные файлы с изображениями, определите, скажем, такой фильтр;

```
"All Image Files|*.bmp;*.gif;*.jpeg;*.jpg;*.jfif;*.png;*.tif;*.tiff|" +
"Windows Bitmap (*.bmp)|*.bmp|" +
"Graphics Interchange Format (*.gif)|*.gif|" +
"JPEG File Interchange Format (*.jpg)|*.jpg;*.jpeg;*.jfif|" +
"Portable Network Graphics (*.png)|*.png|" +
"Tag Image File Format (*.tif)|*.tif;*.tiff|" +
"All Files (*.*)|*.*";
```

В этом случае в комбинированном окне Files Of Type будет показываться семь строк. Как вы заметили, некоторые типы файлов связаны с несколькими шаблонами — в этом случае шаблоны разделяются точкой с запятой. Так, если пользователь выбирает в комбинированном окне строку:

```
JPEG File Interchange Format (*.jpg)
```

то в диалоговом окне показываются файлы с расширениями jpg, jpeg и jfif. Часть, показываемая в диалоговом окне, могла бы иметь вид:

```
JPEG File Interchange Format
```

ИЛИ:

JPEG/JFIF

ИЛИ:

JPEG File Interchange Format (*.jpg, *.jpeg, *.jfif)

Применяйте тот вариант, который, на ваш взгляд, наиболее удобен всего для пользователя.

Следующие два свойства позволяют указать фильтр, используемый в комбинированном окне Files Of Type, и номер первоначально выводимой строки фильтра:

Свойства *FileDialog* (выборочно)

Тип	Свойство	Доступ
string	*Filler*	Чтение/запись
int	*FilterIndex*	Чтение/запись

При возврате из *ShowDialog* свойство *FilterIndex* показывает индекс строки, выбранной пользователем в комбинированном окне Files Of Type. Эти строки нумеруются с 1,

В следующей программе используется *OpenFileDialog*с фильтром, указывающим типы файлов, поддерживаемые статическим методом *Image.FromFile*.

ImageOpen.cs

```
//------------------------------------
// ImageOpen.cs © 2001 by Charles Petzold
//------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class ImageOpen: Form
{
     protected string strProgName;
     protected string strFileName;
     protected Image  image;

     public static void Main()
     {
          Application.Run(new ImageOpen());
     }
     public ImageOpen()
     {
          Text = strProgName = "Image Open";
          ResizeRedraw = true;

          Menu = new MainMenu();
          Menu.MenuItems.Add("&File");
```

см. след. стр.

```
        Menu.MenuItems[0].MenuItems.Add(new MenuItem("&Open...",
                                        new EventHandler(MenuFileOpenOnClick),
                                        Shortcut.Ctrl0));
    }
    void MenuFileOpenOnClick(object obj, EventArgs &a)
    {
        OpenFileDialog dlg = new OpenFileDialog();

        dlg.Filter = "All Image Files|*.bmp;*.ico;*.gif;*.jpeg;*.jpg;" *
                          "*.jfif;*.png;*.tif;*.tiff;*.wmf;*.emf|" +
                     "Windows Bitmap (*.bmp)|*.bmp|" +
                     "Windows Icon (*.ico)|*.ico|" +
                     "Graphics Interchange Format (*.gif)|*.gif|" *
                     "JPEG File Interchange Format (*.jpg)|" +
                          "*.jpg;*.jpeg;*.jfif|" *
                     "Portable Network Graphics (*.png)|*.png|" *
                     "Tag Image File Format (*.tif)|*.tif;*.tiff|" +
                     "Windows Metafile (*.wmf)|*.wmf|" +
                     "Enhanced Metafile (*.emf)|*.emf|" +
                     "All Files (*.*)|*.*";

        if (dlg.ShowDialog() == DialogResult.OK)
        {
            try
            {
                image = Image.FromFile(dlg.FileName);
            }
            catch (Exception exe)
            {
                MessageBox.Show(exc.Message, strProgName); ;
                return;
            }
            strFileName = dlg.FileName;
            Text - strProgName + " - " * Path.GetFileName(strFileName);
            Invalidate();
        }
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

        if (image != null)
            grfx.DrawImage(image, 0, 0);
    }
}
```

Эта программа по структуре аналогична HeadDump за исключением того, что сразу же после возврата управления методом *ShowDialog* она пытается загрузить

изображение. Хотя существование файла проверяется в самом диалоговом окне, ничто не мешает пользователю указать существующий файл, не являющийся файлом изображения. Если он не является файлом изображения (или по каким-то причинам поврежден), *ImageFromFile* генерирует исключение, и программа выводит окно с сообщением о проблеме, В методе *OnPaint* просто используется *DrawImage* для показа загруженного изображения.

Диалоговое окно Save File

Не были ли вы близки к тому, чтобы написать программу, преобразующую различные форматы изображений с помощью стандартных диалоговых окон Open и Save? Здесь есть небольшие тонкости, и моя реализация неоптимальна, но тем не менее остаток главы я посвящу решению этой задачи,

В классе *SaveFileDialog* к свойствам, определенным в *FileDialog*, добавляются:

Свойства *SaveFileDialog*

Тип	Свойство	Доступ	По умолчанию
bool	*CreatePrompt*	Чтение/запись	*false*
bool	*OverwritePrompt*	Чтение/запись	*true*

Эти два свойства влияют на вывод информационных окон, показываемых в то время, когда само окно *SaveFileDialog* еще находится на экране. Если *CreatePrompt* присвоено *true,* а пользователь указал несуществующий файл, то диалоговое окно выведет вопрос, действительно ли пользователь хочет создать этот файл. Если оставить значение *(rue* у свойства *OverwritePrompt,* то диалоговое окно будет запрашивать подтверждение у пользователя, если тот выберет уже существующий файл.

Обычно, когда приложение вызывает диалоговое окно для сохранения документа, приложение предлагает некоторое имя файла и очень часто — расширение по умолчанию. Так, в Notepad в диалоговом окне сохранения показывается шаблон имени файла *.txt,

Однако для программы, выполняющей преобразование изображений из одного формата в другой, правильнее в диалоговом окне сохранения файла предлагать имя файла *без* расширения. Имя файла должно быть таким же, что и у загруженного файла. Но расширение файла должно зависеть от формата, в котором пользователь хочет сохранить файл, поэтому оно неизвестно в момент показа диалогового окна.

Вот каким путем мы пойдем. Во-первых, перед показом диалогового окна присвоим свойству *FileName* класса *SaveFileDialog* имя открытого файла без расширения. Чтобы удалить расширение из имени файла, можно вызвать статический метод *Path.GetFileNameWithoutExtension*. Во-вторых, укажем, что диалоговое окно само должно добавить расширение к выбранному файлу, присвоив значение *true* свойству:

Свойства *FileDialog* (выборочно)

Тип	Свойство	Доступ	По умолчанию
bool	*AddExtension*	Чтение/запись	*false*

Расширение, добавляемое *SaveFileDialog* к имени файла, — это первое расширение файла, указанное в строке фильтра, которую выберет пользователь в комбинированном окне Тип файлов (Save As Type).

Вот одна из возможных реализаций программы. Класс *ImageIO* переопределяет класс *ImageOpen* и добавляет в меню пункт Save As,

ImageIO.cs

```
//-----------------------------------------
// ImageIO.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;

class ImageIO: ImageOpen
{
     MenuItem miSaveAs;

     public new static void Main()
     {
          Application.Run(new ImageIO());
     }
     public ImageIO()
     {
          Text = strProgName = "Image I/O";

          Menu.MenuItems[0].Popup += new EventHandler(MenuFileOnPopup);
          miSaveAs = new MenuItem("Save &As...");
          ffliSaveAs.Click += new EventHandler(MenuFileSaveAsOnClick);
          Menu.MenuItems[0].MenuItems.Add(miSaveAs);
     }
     void MenuFileOnPopup(object obj, EventArgs ea)
     {
          miSaveAs.Enabled = (image != null);
     }
     void MenuFileSaveAsOnClick(object obj, EventArgs ea)
     {
          SaveFileDialog savedlg = new SaveFileDialog();

          savedlg.InitialDirectory = Path.GetDirectoryName(strFileName);
          savedlg.FileName = Path.GetFileNameWithoutExtension(strFileName);
          savedlg.AddExtension = true;
          savedlg.Filter = "Windows Bitmap (*.bmp)|*.bmp|" +
                           "Graphics Interchange Format (*.gif)|*.gif|" +
                           "JPEG File Interchange Format (*.jpg)|" +
                             "*.jpg;*.jpeg;*.jfif|" +
```

см. след. стр.

```
                    "Portable Network Graphics (*.png)|*.png|" +
                    "Tagged Imaged File Format (*.tif)|*.tif;*.tiff";

        ImageFormat[] aif = { ImageFormat.Bmp,  ImageFormat.Gif,
                              ImageFormat.Jpeg, ImageFormat.Png,
                              ImageFormat.Tiff };

        if (savedlg.ShowDialog() == DialogResult.OK)
        {
            try
            {
                image.Save(savedlg.FileName, aif[savedlg.FilterIndex - 1]);
            }
            catch (Exception exc)
            {
                MessageBox.Show(exc.Message, Text);
                return;
            }
            strFileName = savedlg.FileName;
            Text = strProgName + " - " + Path.GetFileName(strFileName);
        }
    }
}
```

Заметьте: в фильтре объекта *SaveFileDialog* нет строк All Files или All Image Files и содержатся только те форматы, что поддерживает метод *Save* класса *Image*. И, цея в том, что пользователь принимает имя файла (без расширения), показываемое в диалоговом окне, и выбирает формат в комбинированном окне Тип файлов (Save As Type). Например, если пользователь выберет «JPEG File Interchange Format», диалоговое окно добавит к базовому имени файла расширение .jpg,

Я уже упоминал, что такая реализация неоптимальна: на мой взгляд, отсутствие расширения у файла может смутить более опытных пользователей. Непонятно, добавит ли программа расширение файла или нет. К счастью, если пользователь введет расширение, диалоговое окно не добавит еще одно расширение.

Как же поступить правильно? Мне нравится, когда имя файла всегда показывается *с расширением* и, если пользователь выбирает различные строки в комбинированном окне Тип файлов (Save As Type), показываемое расширение соответствующим образом изменяется. При программировании с применением Win32 API можно получать доступ к различным элементам управления общих диалоговых окон и использовать приемы такого рода. Однако в интерфейсе Windows Forms доступ к этим элементам управления скрыт от программиста.

Другой подход: пусть пользователь решает, в какой формат он будет преобразовывать изображение, перед показом диалогового окна сохранения файла. Этого можно добиться, создав подменю пункта Save As и перечислив в нем различные форматы: Windows Bitmap, Graphics Interchange Format, JPEG и т. д. Каждый из этих подпунктов будет вызывать *SaveFileDialog* с фильтром из одной строки, соответствующей определенному типу файла.



ГЛАВА

17

Кисти и перья

Кисти и перья — основа графической системы Windows Forms. Перья служат для рисования прямых и кривых, а кисти — для заливки областей, ограниченных прямыми и кривыми, и для рисования текста. Без представления о перьях и кистях начинать работу с Windows Forms очень трудно: если вы не знаете, что такое кисть, то даже не сможете вывести текст.

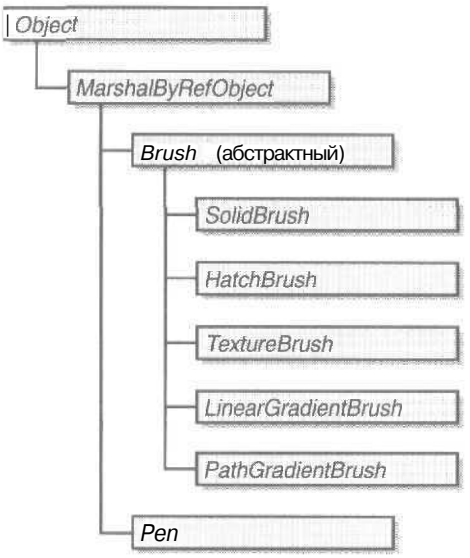
Более того, кисти можно назвать одним из самых всеобъемлющих объектов Windows Forms, поскольку они создаются на основе контуров и растровых изображений, а это далеко не азы программирования графики. Хотя кисти касаются практически всего, что мы узнали до сих пор о программировании графики в Windows Forms, перья можно назвать еще более общими объектами, так как они могут быть созданы на основе кистей,

Схему иерархии классов, которые мы обсудим в этой главе, см. ниже.
Классы *Brush* и *Pen* определяются в пространстве имен *System.Drawing*, но *Brush* — это абстрактный класс. поэтому создать его экземпляр нельзя. Два из пяти классов, производных от *Brush* — *SolidBrush* и *TextureBrush* — определяются в пространстве имен *System.Drawing*, а *HatchBrush*, *LinearGradientBrush* и *PathGradientBrush* — в пространстве имен *System.Drawing.Drawing2D*.

Поскольку неабстрактные классы. определяющие кисти, наследуют классу *Brush*, зачастую удобно хранить их экземпляры в переменной типа *Brush*. В этой книге я часто прибегал к такому приему:

```
Brush brush = new SolidBrush(ForeColor);
```

Однако классы-потомки *Brush* определяют собственные изменяемые свойства, с которыми проще работать, если сохранить объект в переменной его собственного типа.



Заливка сплошными цветами

Класс *SolidBrush* — несомненно, самый простой среди потомков *Brush*. У него имеется единственный конструктор, который мы используем с главы 3:

Конструктор *SolidBrush*

```
SolidBrush (Color color)
```

и единственное свойство:

Свойство *SolidBrush*

Тип	Свойство	Доступ
Color	*Color*	Чтение/запись

Обычно, когда требуется кисть для заливки сплошным цветом, лучше задействовать возможности класса *Brushes*. Каждое из его статических неизменяемых свойств, которых у него 141, возвращает объект типа *Brush*, основанный на одном из стандартных цветов.

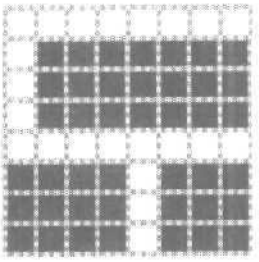
Узорные кисти

Узорная кисть заливает область орнаментом, основанным на небольшом повторяющемся шаблоне, который чаще всего состоит из горизонтальных, вертикальных или диагональных линий. Узорная кисть может показаться «антиквариатом* компьютерной графики, оживляющим воспоминания об ушедшей эпохе черно-белых гистограмм и круговых диаграмм и других столпов деловой графики.

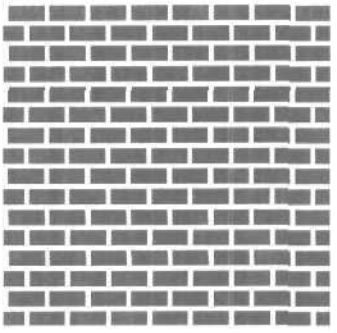
Хотя с появлением возможности использования цвета потребность в узорных кистях заметно снизилась, они до сих пор играют важную роль в выводе графи-

ки. Множество пользователей (включая меня) несмотря ни на что предпочитают черно-белые лазерные принтеры цветным струйным. Иногда, например, при печати цветных фотографий на черно-белом принтере удается получить удовлетворительные результаты, сопоставляя цвета оттенкам серого. Но бывает, разные цвета используют для представления данных, например, для обозначения на карте численности населения. Вид подобных цветных графиков сильно искажается, когда визуализация осуществляется при помощи механического сопоставления цветов оттенкам серого. В этом случае узорная кисть позволяет получить более удобочитаемые карты.

В Windows Forms шаблоны узоров основаны на монохромных битовых картах в виде квадрата размером 8x8 пикселов. Вот пример такого шаблона для узора «кирпичная кладка»:



Этот шаблон станет похожим на кирпичную кладку, только если размножить его по вертикали и горизонтали:



Класс *HatchBrush* определяется в пространстве имен *System.Drawing.Drawing2D*. У этого класса лишь два конструктора:

Конструкторы *HatchBrush*

```
HatchBrush(HatchStyle hs, Color clrForeground)
HatchBrush(HatchStyle hs, Color clrForeground, Color clrBackground)
```

Хотя для создания узорных кистей применяются монохромные битовые карты, нули и единицы такой карты могут быть сопоставлены любой паре цветов. В более простых узорных кистях, заливающих области линейной штриховкой, цвет изображения соответствует цвету линий штриховки, а цвет фона — цвету промежутков между линиями.

Будьте внимательны: поначалу при использовании этого конструктора часто выбирают черный цвет изображения, **скажем,** так:

```
new HatchBrush(hs, Color.Black)  // Будет работать неправильно!
```

или так:

```
new HatchBrush(hs, ForeColor)    // Скорее всего, будет работать неправильно!
```

Проблема в том, что по умолчанию именно цвет *фона* является черным, поэтому следует выбрать для изображения белый цвет, чтобы получить максимально контрастную КИСТЬ:

```
new HatchBrush(hs, Color.White)  // Верно!
```

Вот эквивалентное выражение более явного вида:

```
new HatchBrush(hs, Color.White, Color.Black)  // Верно!
```

Большинство узорных кистей прекрасно выглядят, если поменять цвета изображения и фона, например так;

```
new HatchBrush(hs, Color.Black, Color.White)  // Удовлетворительно
```

Однако в этом случае узор кирпичной кладки выглядит немного необычно: кирпичи получаются черными, а швы между ними — белыми.

Объект *HatchBrush* полностью определяется тремя свойствами, соответствующими трем аргументам второго конструктора:

Свойства *HatchBrush*

Тип	Свойство	Доступ
HatcbStyle	*HatcbStyle*	Чтение
Color	*ForegroundColor*	Чтение
Color	*BackgroundColor*	Чтение

Значение свойства *HatcbStyle* — это член перечисления *HatcbStyle.* Реализация узорных кистей в GDI+ из Windows Forms отличается от таковой в Windows GDI большим числом стилей узора. Если в GDI всего 6 видов штриховки (горизонтальная, вертикальная, пара диагональных и два вида штриховки крест-накрест), то в GDI+ их 53.

Перечисление *HatchStyle* содержит по одному члену для каждого из 53 стилей (в документации описаны 54, но *Cross* и *LargeGrid* идентичны), а также пару членов, облегчающих использование значений перечисления, избавляя от необходимости жестко программировать их:

Перечисление *HatchStyle* (выборочно)

Член	Значение
Mm	0
Max	**4**

Теоретически, исходя из числа членов перечисления, можно предположить существование 53 членов, пронумерованных от 0 до 52. К сожалению, для *HatchStyleMax*

непонятным образом вместо ожидаемого номера 52 задан номер 4. Если бы удалось исправить это и назначить для *HatchStyleMax* истинный максимальный номер, жить стало бы гораздо легче за счет повышения степени абстрактности кода, представляющего пользователю все варианты стиля узорной кисти. Мы рассмотрим пару программ, в которых максимальное и минимальное значения перечисления *HatchStyle* запрограммированы жестко.

Следующая программа выводит прямоугольники размером 32x32 пиксела, демонстрирующие все стили узора,

HatchBrushArray.cs

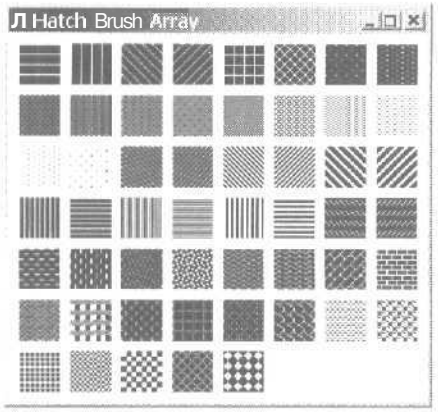
```
//---------------------------------------------
// HatchBrushArray.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HatchBrushArray: PrintableForm
{
     const int iSize = 32, iMargin = 8;
     const int iMin = 0, iMax = 52;  // Минимальное и максимальное значения
HatchStyle

     public new static void Main()
     {
          Application.Run(new HatchBrushArray());
     }
     public HatchBrushArray()
     {
          Text = "Hatch Brush Array";
          ClientSize = new Size(8 * iSize + 9 * iMargin,
                                1 * iSize + 8 * iMargin);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          for (HatchStyle hs = (HatchStyle)iMin; hs <= (HatchStyle)iMax;
               hs++)
          {
               HatchBrush hbrush =
                              new HatchBrush(hs, Color.White, Color.Black);
               int y = (int)hs / 8;
               int x = (int)hs % 8;

               grfx.FillRectangle(hbrush, iMargin + x * (iMargin + iSize),
                                          iMargin + y * (iMargin + iSize),
                                          iSize, iSize);
          }
     }
}
```

На экране результат программы выглядит так:



Поскольку прямоугольники здесь являются квадратами со стороной 32 пиксела, орнамент в каждом из них образован 16 квадратами 8x8 пикселов, расположенными в 4 ряда по 4 квадрата.

Можно распечатать шаблоны узора, щелкнув клиентскую область программы. Визуализация шаблонов на разных принтерах может немного различаться. Так, на моем лазерном принтере каждый элемент орнамента размером 8×8 пикселов выводится в виде квадрата со стороной 1/15 дюйма. Преобразование страницы, заданное по умолчанию для принтера, делает его устройством с кажущимся разрешением 100 точек на дюйм. В результате квадратные прямоугольники со стороной 32 пиксела, выводимые программой HatchBrushArray. печатаются как квадраты со стороной 0,32 дюйма. Это позволяет уместить в таком квадрате 25 копий шаблона узора, расположенных в 5 рядов по 5 копий в каждом. Однако, когда я направляю выводимую графику па факс, в каждом прямоугольнике оказывается по 64 копии шаблона,

Преобразования не влияют на шаблон узора! Если вставить в HatchBrushArray операторы, назначающие преобразование страницы, отличное от заданного по умолчанию, окажется. что преобразования влияют на положение и размер выводимых прямоугольников (что естественно), но не влияют на внешний вид шаблона. Линии орнамента не становятся шире и не приближаются друг к другу не происходит и их вращения или сдвига.

В графическом редакторе очень кстати была бы возможность выбора стиля узора, удобная для пользователя. Вот программа, которая размещает в меню все стили узора,

HatchBrushMenu.cs

```
//-----------------------------------------
// HatchBrushMenu.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
```

см. след. стр.

```
using System.Windows.Forms;

class HatchBrushMenu: Form
{
     HatchStyleMenuItem hsmiChecked;
     const int iMin = 0, iMax = 52;  // Минимальное и максимальное значения
HatchStyle

     public static void Main()
     {
          Application.Run(new HatchBrushMenu());
     }
     public HatchBrushMenu()
     {
          Text = "Hatch Brush Menu";
          ResizeRedraw = true;

          Menu = new MainMenu();
          Menu.MenuItems.Add("&Hatch-Style");

          for (HatchStyle hs = (HatchStyle)iMin; hs <= (HatchStyle)iMax;
               hs++)
          {
               HatchStyleMenuItem hsmi - new HatchStyleMenuItem();

               hsmi.HatchStyle = hs;
               hsmi.Click += new EventHandler(MenuHatchStyleOnClick);

               if ((int)hs X B == 0)
                    hsmi.BarBreak = true;

               Menu.MenuItems[0].MenuItems.Add(hsmi);
          }
          hsmiChecked - (HatchStyleMenuItem) Menu.MenuItems[0].MenuItems[0];
          hsmiChecked.Checked = true;
     }
     void MenuHatchStyleOnClick(object obj, EventArgs ea)
     {
          hsmiChecked.Checked = false;
          hsmiChecked = (HatchStyleMenuItem) obj;
          hsmiChecked.Checked = true;

          Invalidate();
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx = pea.Graphics;

          HatchBrush hbrush = new HatchBrush(hsmiChecked.HatchStyle,
                                             Color.White, Color.Black);
```

см. след. стр.

```
            grfx.FillEllipse(hbrush, ClientRectangle);
        }
I
class HatchStyleMenuItem: MenuItem
{
    const    int cxImage - 32, cyImage = 32, iMargin = 2;
    readonly int cxCheck, cyCheck;

    public HatchStyle HatchStyle;

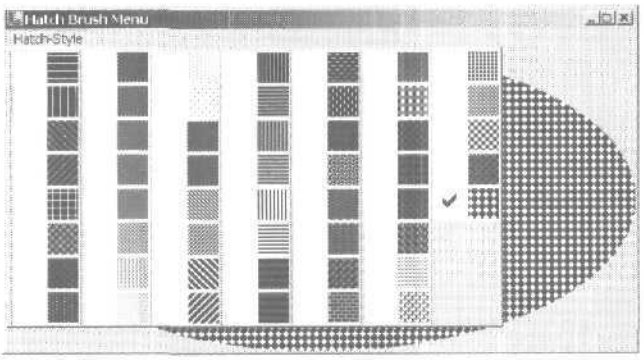
    public HatchStyleMenuItem()
    {
        OwnerDraw = true;

        cxCheck = SystemInformation.MenuCheckSize.Width;
        cyCheck = SystemInformation.MenuCheckSize.Height;
    }
    protected override void OnMeasureItem(MeasureItemEventArgs miea)
    {
        miea.ItemWidth  = 2 * cxImage + 3 - iMargin - cxCheck;
        miea.ItemHeight =     cyImage + 2 * iMargin;
    }
    protected override void OnDrawItem(DrawItemEventArgs diea)
    {
        diea.DrawBackground();

        if ((diea.State & DrawItemState.Checked) != 0)
        <
            ControlPaint.DrawMenuGlyph(diea.Graphics,
                                       diea.Bounds.Location.X + iMargin,
                                       diea.Bounds.Location.Y + iMargin,
                                       cxImage, cylmage, MenuGlyph.Checkmark);
        )
        HatchBrush hbrush = new HatchBrush(HatchStyle,
                                           Color.White, Color.Black);

        diea.Graphics.FillRectangle(hbrush,
                                    diea.Bounds.X + 2 * iMargin + cxImage,
                                    diea.Bounds.Y + iMargin, cxlmage, cylmage);
    }
}
```

Эта программа определяет класс *HatchStyleMenuItem* являющийся подклас сом *MenuItem*, необходимый не только для реализации поля *HatchStyle*, но и для юддержки элементов базового класса. Когда приложение создает это меню, оно вс гавляет разделители через каждые 8 элементов. В результате вложенное меню оказывается достаточно большим, но не настолько, чтобы затруднить пользоват елю выбор стиля узора:



Пора познакомиться с реальными именами стилей узора. Для изображений, показанных в приведенных ниже таблицах, задан черный цвет фона по умолчанию и белый цвет изображения. Каждый прямоугольник образован 16 копиями шаблона узора (расположенных в 4 ряда по 4 копии в каждом).

Эти шесть значений перечисления *HatchStyle* имеют аналоги в GDI Windows:

Перечисление *HatchStyle* (выборочно)

Член	Значение	Вид
Horizontal	0	
Vertical	1	
ForwardDiagonal	2	
BackwardDiagonal	3	
Cross или LargeGrid	4	
DiagonalCross	5	

Следующие стили имитируют оттенки серого. Имена членов отражают приблизительный процент заполнения шаблона узора цветом изображения:

Перечисление *HatchStyle* (выборочно)

Член	Значение	Вид
Percent05	6	
Percent10	7	
Percent20	8	
Percent25	9	
Percent30	10	
Percent40	11	
Percent50	12	
Percent60	13	
Percent70	14	
Percent75	15	

Перечисление *HatchStyle* (выборочно) *(продолжение)*

Член	Значение	Вид
Percent80	16	
Percent90	17	

Далее идут вариации стандартных видов штриховки: горизонтальная, вертикальная и диагональная:

Перечисление *HatchStyle* (выборочно)

Член	Значение	Вид
LightDownwardDiagonal	18	
LightUpwardDiagonal	19	
DarkDownwardDiagonal	20	
DarkUpwardDiagonal	21	
WideDownwardDiagonal	22	
WideUpwardDiagonal	23	
LightVertical	24	
LightHorizontal	25	
NarrowVertical	26	
NarrowHorizontal	27	
DarkVertical	28	
DarkHorizontal	29	
DashedDownwardDiagonal	30	
DashedUpwardDiagonal	31	
DashedHorizontal	32	
DashedVertical	33	

И завершает перечисление группа разнообразных шаблонов:

Перечисление *HatchStyle* (выборочно)

Член	Значение	Вид
SmallConfetti	34	
LargeConfetti	35	
ZigZag	36	
Wave	37	
DiagonalBrick	38	
HorizontalBrick	39	
Weave	40	
Plaid	41	

Перечисление *HatchStyle* (выборочно) *(продолжение)*

Член	Значение	Вид
Divot	42	
DottedGrid	43	
DottedDiamond	44	
Shingle	45	
Trellis	46	
Sphere	47	
SmallGrid	48	
SmallCheckerBoard	49	
LargeCheckerBoard	50	
OutlinedDiamond	51	
SolidDiamond	52	

Начальная точка визуализации узора

Об узорной кисти следует знать кое-что еще, что относится и к более сложным видам кистей. Фактически речь идет о составляющей части модели, которая легла в основу графической системы Windows Forms. Чтобы понять, что я имею в виду, рассмотрим такую программу.

OverlappingHatchBrushes.cs

```
//-------------------------------------------------
// OverlappingHatchBrushes.cs e 2001 by Charles Petzold
//- ---------------------------------- - --------.---  --
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class OverlappingHatchBrushes: PrintableForm
{
     public new static void Main()
     {
          Application flun(new OverlappingHatchBrushes());
     }
     public OverlappingHatchBrushes()
     {
          Text = "Overlapping Hatch Brushes";
     \
     protected override void DoPage(Graphics grfx, Color clr, int cx, int ey)
     {
          HatchBrush hbrush = new HatchBrush(HatchStyle.HorizontalBrick,
```

см. след. стр.

```
                                                Color.White);
            for (int i = 0; i < 10; i++)
                grfx.FillRectangle(hbrush, i * cx / 10, i * cy / 10,
                                           cx / 8, cy / 8);
        }
    }
```

Эта программа рисует одной узорной кистью 10 перекрывающихся прямоугольников. Однако между ними, похоже, нет четких границ, так как узор как бы «перетекает» из одного прямоугольника в другой;



При заливке области узором его шаблон просто размножается по горизонтали и по вертикали, но это еще не все. Шаблон изначально должен быть ориентирован относительно пиксела с определенными координатами. Можно подумать, что шаблон ориентируется относительно рисуемого графического объекта, например, левого верхнего угла прямоугольника. Но эта программа совершенно четко демонстрирует, что все совсем *не так*. Фактически битовая карта шаблона ориентируется относительно верхнего левого угла области, в которой осуществляется рисование, т. е. относительно верхнего левого *угла клиентской* области или области печати страницы принтера.

Это легко представить, если вообразить кисть, покрывающую всю поверхность, на которую ссылается объект *Graphics*. Заливая область кистью, вы на самом деле как бы делаете «прорезь» в «трафарете», через который видна поверхность узорной кисти.

Можно изменить ориентацию узорной кисти, заданную по умолчанию, используя свойство класса *Graphics*:

Свойства *Graphics* (выборочно)

Тип	Свойство	Доступ
Point	*RenderingOrigin*	Чтение/запись

Оно влияет только на узорные кисти (для изменения ориентации других типов кистей существуют иные подходы). По умолчанию значением *RenderingOrigin* является точка с координатами (0. 0). Если задать другую точку (в координатах на устройстве), то в дальнейшем все узорные кисти будут ориентированы относительно этой точки.

Вот программа, рисующая 10 выстроенных по диагонали прямоугольников, для каждого из которых задано собственное значение *RenderingOrigin*.

HatchBrushRenderingOrigin.cs

```
//-----------------------------------------------
// HatchBrushRenderingOrigin.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HatchBrushRenderingOrigin : PrintableForm
{
     public new static void Main() ,
     <
          Application.Run(new HatchBrushRenderingOrigin());
     }
     public HatchBrushRenderingOrigin()
     {
          Text = "Hatch Brush Rendering Origin";  .
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy):
     {
          HatchBrush hbrush = new HatchBrush(HatchStyle.HorizontalBrick,
                                             Color.White);
          for (int i = 0; i < 10; i++)
          {
               grfx.RenderingOrigin - new Point(i * ex / 10, i * cy / 10);

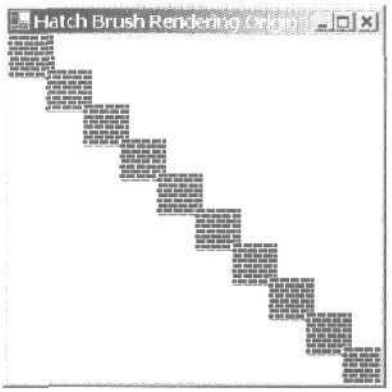
               grfx. FillRectangle(hbrush, i * ex / 10, i * ey / 10,
                                           cx / 8, cy / 8);
          }
     }
}
```

В результате исполнения программы получается картинка, демонстрирующая, что в левом верхнем углу каждого прямоугольника прорисовка узора начинается заново,

Можно привести пару ситуаций, в которых приходится изменять начальную точку визуализации узора. Когда узорная кисть используется для рисования столбчатых диаграмм, обычно все шаблоны узоров ориентируются относительно левого верхнего угла области рисования. При этом, даже если используются разные узоры, соседние столбцы могут сливаться, и результат получается малопривлекательным. В таком случае для каждого столбца надо задавать собственную начальную точку визуализации узора, чтобы сориентировать шаблон узора относительно верхнего левого угла заливаемого столбца,

Может встать и обратная задача. Допустим, родительское окно и одно из его дочерних заливают одинаковым узором. При этом надо сделать так, чтобы дочернее

окно слилось с родительским. Но узор в дочернем окне ориентирован относительно его левого верхнего угла, поэтому для начальной точки визуализации узора в дочернем окне нужно задать координаты левого верхнего угла родительского окна, дабы получить желаемый результат.



Текстурные кисти

Если, просмотрев весь список узорных кистей, вы так и не нашли подходящей, нужна кисть, сочетающая более двух цветов, или необходимо, чтобы кисть была подвержена преобразованиям, попробуйте задействовать класс *TextureBrush*.

Текстурная кисть создается на базе объекта типа *Image* или прямоугольной части этого объекта, размноженной по горизонтали и вертикали.

Конструкторы *TextureBrush*

```
TextureBrush(Image image)
TextureBrush(Image image, Rectangle rectSrc)
TextureBrush(Image image, RectangleF rectfSrc)
TextureBrush(Image image, WrapMode wm)
TextureBrush(Image image, WrapMode wm, Rectangle rectSrc)
TextureBrush(Image image, WrapMode wm, RectangleF rectfSrc)
TextureBrush(Image image, Rectangle rectSrc, ImageAttributes ia)
TextureBrush(Image image, RectangleF rectSrc, ImageAttributes ia)
```

Перечисление *WrapMode* определяется в пространстве имен *System.Drawing.Drawing2D*:

Перечисление *WrapMode*

Член	Значение
Tile	0
TileFlipX	1
TileFlipY	2
TileFlipXY	3
Clamp	4

Как будет показано ниже, это перечисление определяет способ размещения текстуры по горизонтали и вертикали. Две версии конструктора с аргументом *Image-Attnbutes* не требуют аргумента *WrapMode,* поскольку у *ImageAttnbutes* есть собственный метод, устанавливающий способ размещения текстуры.

Вот программа, которая создает текстурную кисть на основе части изображения космического корабля Аполлон 11 (см. главу 11). Команда меню позволяет менять значение *WrapMode.* Метод *OnPaint* выводит два перекрывающихся эллипса, залитых текстурной кистью,

TextureBrushDemo.cs

```
//--------------------------------------------
// TextureBrushDemo.cs © 2001 by Charles Petzold
//--------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TextureBrushDemo: PrintableForm
{
     MenuItem     miChecked;
     TextureBrush tbrush;

     public new static void Main()
     {
          Application.Run(new TextureBrushDemo());
     }
     public TextureBrushDemo()
     {
          Text = "Texture Brush Demo";

          Image iaage = Image.FromFile(
               "..\\..\\..\\..\\Images and Bitmaps\\Apollo11FullColor.jpg");

          tbrush = flew TextureBrush(image, new Rectangle(95, 0, 50, 55»;

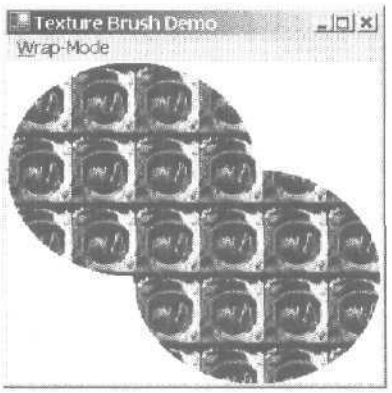
          Menu = new MainMenu();
          Menu.MenuItems.Add("&Wrap-Mode");

          foreach (WrapMode wm in Enum.GetValues(typeof(WrapMode)))
          {
               MenuItem mi = new MenuItem();
               mi.Text     = wm.ToString();
               mi.Click   += new EventHandler(MenuWrapModeOnClick);
               Menu.MenuItems[0].MenuItems.Add(mi);
          }
          miChecked = Menu.MenuItems[0].MenuItems[0];
          raiChecked.Ghecl«d-true;
     }
```

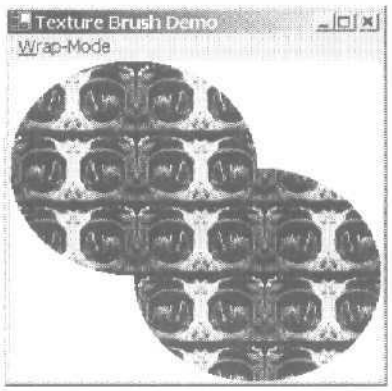
см. след. стр.

```
    void MenuWrapModeOnClick(object obj, EventArgs ea)
    {
        miChecked.Checked = false;
        miChecked = (MenuItem) obj;
        miChecked.Checked = true;
        tbrush.WrapMode = (WrapMode)miChecked.Index;
        Invalidate();
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.FillEllipse(tbrush, 0, 0, 2 * cx / 3, 2 * cy / 3);
        grfx.FillEllipse(tbrush, cx / 3, cy / 3, 2 * cx / 3, 2 * cy / 3);
    }
}
```

Помимо прочего, программа демонстрирует эффект «взгляда сквозь трафарет», характерный для графики Windows Forms. Поскольку метод *OnPaint* использует одну и ту же кисть для заливки обеих фигур, рисунок внутри эллипсов в точности совпадает:



По умолчанию значение *WrapMode* равно *Tile*, что определяет повтор по горизонтали и вертикали идентичных фрагментов изображения. Если с помощью меню поменять значение *WrapMode* на *TileFlipX*, то изображения в соседних столбцах будут отражены по вертикали:



Если для *WrapMode* задать *TileFlipY*, то изображения в соседних рядах будут перевернуты. Значение *TileFlipXY* комбинирует оба эффекта.

У значения *Clamp* самый экстремальный эффект: оно определяет кисть из одного-единственного фрагмента, расположенного в левом верхнем углу клиентской области (его правый нижний угол виден в верхнем эллипсе), а остальная часть кисти прозрачна:



Как я говорил, преобразование страницы и глобальное преобразование не действуют на узорные кисти. Однако они влияют на текстурные кисти.

Допустим, в программе TextureBrushDemo модифицировали метод *DoPage*. Измененный метод рисует не эллипс, а прямоугольник, в котором точно умещаются 9 копий изображения размером 50x55 пикселов, например:

```
grfx.FillRectangle(tbrush, 0, 0, 150, 165);
```

Не важно, какие параметры будут заданы для преобразования страницы и глобального преобразования. Независимо от параметров масштабирования, сдвига или вращения результирующий параллелограмм (в общем случае прямоугольник будет нарисован как параллелограмм) всегда будет заполнен девятью копиями шаблона, которые будут масштабироваться, сдвигаться и вращаться соответствующим образом.

Выше я уже сказал, что свойство *RenderingOrigin* объекта *Graphics* влияет только на объект *HatchBrush*. Альтернативное свойство класса *TextureBrush* позволяет воспроизвести эффект свойства *RenderingOrigin* и сделать много чего еще. Вот полный список свойств объекта *TextureBrush*:

Свойства *TextureBrush*

Тип	Свойство	Доступ
Image	*Image*	Чтение
WrapMode	*WrapMode*	Чтение/запись
Matrix	*Transform*	Чтение/запись

Верно, третье свойство — это матричное преобразование, действующее на саму кисть. Например, если добавить к методу *DoPage* программы TextureBrushDemo оператор:

```
tbrush.Transform = new Matrix(2, 0, 0, 2, 0, 0);
```

размер и положение эллипсов (прямоугольников или любых других фигур, которые рисует этот метод) не изменятся, но изображения, заполняющие фигуру, будут увеличены вдвое до размера 100×110 пикселов. Эти изображения можно также наклонять и вращать. Кроме того, в классе *TextureBrush* есть методы *TranslateTransform*, *ScaleTransform*, *RotateTransform*, *MultiplyTransform* и *ResetTransform*, весьма похожие на одноименные методы класса *Graphics* (это единственные методы объекта *TextureBrush*, не имеющие аналогов у *Brush*). Трансляция позволяет имитировать перемещение начальной точки визуализации текстуры.

Подобно одноименным методам класса *Graphics*, эффект методов класса *TextureBrush*, определяющих преобразования, является кумулятивным. Скажем, если поместить в метод *DoPage* оператор:

```
tbrush.RotateTransform(45);
```

при каждом вызове этого метода текстура кисти будет дополнительно поворачиваться на 45°. Чтобы избежать непредсказуемых результатов, надо вызывать метод *RotateTransform* перед *RotateTransform* или, что еще лучше, поместить вызов *RotateTransform* в конструктор программы. Вот что при этом получается:

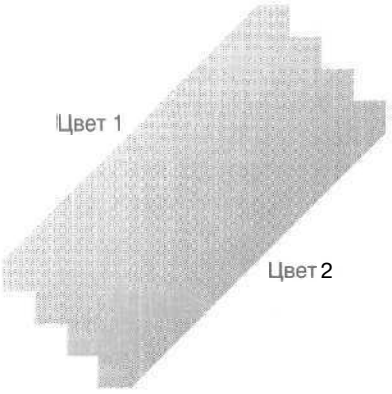


Свойство *Transform* класса *TextureBrush* влияет только на размер и ориентацию текстуры, но не на объекты, рисуемые этой кистью. Напротив, одноименное свойство объекта *Graphics* влияет на размер и ориентацию как текстуры кисти, так и всех рисуемых объектов. Если определить свойство *Transform* у обоих объектов, текстура кисти подвергнется составному преобразованию.

Кисти с линейным градиентом

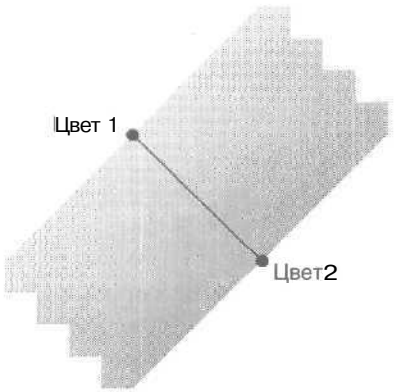
Нам остается рассмотреть два класса: *LinearGradientBrush* и *PathGradientBrush*. Слово «gradient» (градиент) в названии этих классов обозначает плавный переход от одного цвета к другому Класс *LinearGradientBrush* определяет взаимный переход цветов, также иногда называемый *фонтанным*. Определение способа перехода от одного цвета к другому может показаться нелегкой задачей. Вероятно, поэтому существуют разные способы определения градиентной кисти.

Градиентный переход можно определить, используя пару параллельных прямых, нарисованных чистыми цветами. Цвет изменяется в пространстве, ограниченном этими прямыми. Вот пример перехода между цветами *Color.LightGray* и *Color.DarkGray*:



Таким образом, кисть с линейным градиентом представляет собой бесконечно длинную полосу, ограниченную параллельными прямыми двух цветов,

Чтобы определить такую кисть, вовсе не обязательно задавать параллельные прямые — намного проще задать пару точек. Отрезок, соединяющий эти точки, пересекает параллельные границы области перехода под прямым углом



Заметьте: существует бесконечное множество пар точек, задающих один и тот же линейный градиент. Я буду называть линию, соединяющую подобную пару точек *линией градиента*. Также введем термин *линия производного цвета* (mix line), которым обозначим линию, проведенную под прямым углом к линии градиента и параллельную граничным линиям.

У класса *LinearGradientBrush* восемь конструкторов. Два из них позволяют задать пару точек и пару цветов:

Конструкторы *LinearGradientBrush* (выборочно)

```
LinearGradientBrush(Point pt1, Point pt2, Color clr1, Color clr2)
LinearGradientBrush(PointF ptf1, PointF ptf2, Color clr1, Color clr2)
```

Единственное различие этих конструкторов в том, что один использует структуры *Point,* а другой — *PointF.* Точки задают в глобальных координатах. Цвет первой точки (*pt1* или *ptf1*) задает аргумент *clr1*, цвет второй *(pt2* или *ptf2*) — *clr2.*

Взгляните на следующую программу. Ее метод *DoPage* создает объект *LinearGradientBrush,* определяющий первую точку как *(cx/4, cy/4),* вторую — как (3*cx/4, 3*cy/4), а соответствующая пара цветов — *Color. White* и *Color Black.* Затем программа закрашивает этой кистью прямоугольник, размер которого совпадает с размером клиентской области.

TwoPointLinearGradientBrush.cs

```
//---------------------------------------------------------
// TwoPointLinearGradientBrush.cs © 2001 by Charles Petzold
//---------------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TwoPointLinearGradientBrush: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new TwoPointLinearGradientBrush());
     }
     TwoPointLinearGradientBrush()
     {
          Text = "Two-Point Linear Gradient Brush";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          LinearGradientBrush lgbrush =
               new LinearGradientBrush(
                         new PoInt(cx / 4, cy / 4),
                         new Point(3 * cx / 4, 3 * cy / 4),
                         Color.White, Color.Black);

          grfx.FillRectangle(lgbrush, 0, 0, cx, cy);
     }
}
```

Пока я ничего не сказал о том. что происходит за пределами полосы, определяемой объектом *LinearGradientBrush.* Как видите, по умолчанию эти области закрашиваются той же кистью:

Ширина непрерывной полосы, протянувшейся из левого нижнего в правый верхний угол прямоугольника, задана координатами точек, определяющих кисть. За пределами полосы (здесь это области в левом верхнем и правом нижнем углах прямоугольника) заливка градиентной кистью повторяется.



Способом заливки управляет свойство *WrapMode* кисти. Значение *WrapMode.Tile* (заданное по умолчанию) эквивалентно *WrapMode.TileFlipY* и вызывает повторную заливку идентичной кистью без отражения (как показано на рис. выше).

Значение *WrapMode.TileFlipX* эквивалентно *Wrapmode.TileFlipXY* и вызывает зеркальное отражение кисти. В результате получается непрерывный переход цветов:



Параметр *WrapMode.Clamp* недопустим для градиентных кистей.

Еще раз подчеркну: любой метод *Fill,* вызываемый с кистью, формирует «окно», через которое видна поверхность кисти. При использовании текстурных и градиентных кистей вид любой залитой области в некоторой степени зависит от *расположения* этой области. Если нарисовать небольшой прямоугольник при помощи кисти, определяемой программой TwoPointLinearGradientBrush, результат будет мало похож на градиент.

Обычно конкретную градиентную кисть определяют на основе реальных координат заливаемого объекта. Так, чтобы залить градиентной кистью прямоугольник, нужно определить кисть с теми же координатами, что были заданы для рисования прямоугольника. При этом удобны конструкторы объекта *LinearGradientBrush* с аргументом-прямоугольником:

Конструкторы *LinearGradientBrush* (выборочно)

```
LinearGradientBrush(Rectangle rect, Color clr1, Color clr2, LinearGradientMode lgm)
LinearGradientBrush(RectangleF rectf, Color clr1, Color clr2, LinearGradientMode lgm)
```

Перечисление *LinearGradientMode* определяет ориентацию линии градиента - пройдет ли она между серединами противоположных сторон или между противоположными углами прямоугольника:

Перечисление *LinearGradientMode*

Член	Значение	Описание
Horizontal	0	Линия градиента проходит горизонтально, цвет *clr1* находится слева, а *clr2* — справа от нее.
Vertical	1	Линия градиента проходит вертикально, цвет *clr1* находится сверху, а *clr2* — снизу от нее,
ForwardDiagonal	2	Линия производного цвета проходит из правого верхнего в левый нижний угол: цвет *clrl* находится в левом верхнем углу, а *clr2* — в правом нижнем.
BackwardDiagonal	3	Линия производного цвета проходит из левого верхнего в правый нижний угол; цвет *clrl* находится в правом верхнем углу, а *clr2* — в левом нижнем.

Обратите внимание на два последних значения перечисления: противоположные углы прямоугольника определяют линию производного цвета, а не линию градиента. Хотя в двух других углах прямоугольника расположены чистые цвета, а граничные линии проходят через вершины этих углов, точки вершин *не* определяют линию градиента, если прямоугольник не является квадратом. Давайте разберем этот случай.

Следующая программа определяет градиентную кисть на основе прямоугольника, высота и ширина которого составляют половину высоты и ширины клиентской области программы. Прямоугольник центрирован в клиентской области программы. Меню позволяет задавать аргумент *LinearGradientMode* конструктора. После заливки клиентской области заданной кистью метод *DoPage* рисует прямоугольник, использованный для создания кисти.

RectangleLinearGradientBrush.cs

```
//  .....  ----------  .....  -------
// ftectangleLirrearGradlent8rush.es © 2001 by Charles Petzold
//----------  ----------  ---------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class RectangleLinearGradientBrush: PrintableForm
{
     MenuIten» miChecked;

     public new static void Main()
     {
          Application.Run(new fiectangleLinearGradientBrush<));.
     }
     public RectangleLinearGradientBrush()
```

см. след. стр.

```
    {
        Text = "Rectangle Linear-Gradient Brush";

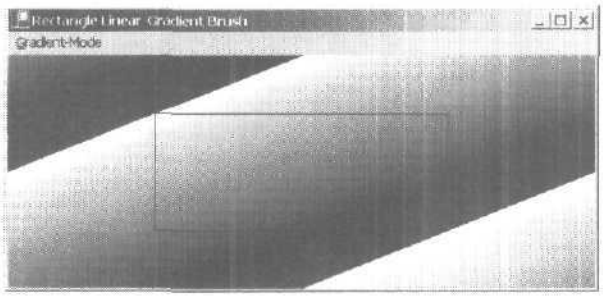
        Menu = new MainMenu();
        Menu.MenuItems.Add("&Gradient-Mode");

        foreach (LinearGradientMode gm in
                        Enum.GetValues(typeof(LinearGradientMode)))
        {
            MenuItem mi = new MenuItem();
            mi.Text     = gm.ToString();
            mi.Click   += new EventHandler(MenuGradientModeOnClick);
            Menu.MenuItems[0].MenuItems.Add(mi);
        }
        miChecked = Menu.MenuItems[0].MenuItems[0];
        miChecked.Checked = true;
    }
    void MenuGradientModeOnClick(object obj, EventArgs ea)
    {
        miChecked.Checked = false;
        miChecked = (MenuItem) obj;
        miChecked.Checked = true;
        Invalidate();
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Rectangle rectBrush =
                        new Rectangle(cx / 4, cy / 4, cx / 2, cy / 2);

        LinearGradientBrush lgbrush =
            new LinearGradientBrush(
                        rectBrush, Color.White, Color.Black,
                        (LinearGradientMode) miChecked.Index);

        grfx.FillRectangle(lgbrush, 0, 0, cx, cy);
        grfx.DrawRectangle(Pens.Black, rectBrush);
    }
}
```

Вот пример, когда в результате выбора в меню значения *LinearGradientMode.ForwardDiagonal*, окно стало шире:



Хотя левый верхний угол прямоугольника, определяющего кисть, закрашен первым цветом, а нижний правый — вторым, видно, что линия градиента не совпадает с линией, проведенной из левого верхнего угла к правому нижнему, поскольку линия градиента всегда проходит под прямым углом к граничным линиям, а линия производного цвета (параллельная граничным линиям и равноудаленная от них) — из правого верхнего в левый нижний угол прямоугольника.

Четыре последних конструктора *LinearGradientBrush* позволяют задавать прямоугольник и угол:

Конструкторы *LinearGradientBrush* (выборочно)

```
LinearGradientBrush (Rectangle rect, Color clr1, Color clr2, float fAngle)
LinearGradientBrush (Rectangle rect, Color clr1, Color clr2, float fAngle, bool bScale)
LinearGradientBrush (RectangleF rectf, Color clr1, Color clr2, float fAngle)
LinearGradientBrush (RectangleF rectf, Color clr1, Color clr2, float fAngle,
                     bool bScale)
```

Если аргумент *fAngle* равен 0, результат идентичен применению *LinearGradientMode.Horizontal*: линия градиента проходит горизонтально от левой стороны прямоугольника к правой.

Если значение *fAngle* увеличивается, линия градиента поворачивается по часовой стрелке на соответствующее число градусов. Первый цвет находится в левом верхнем углу прямоугольника, а второй — в правом нижнем. Когда *fAngle* достигает 90°, эффект становится идентичным эффекту значения *LinearGradientMode.Vertical*. При этом линия градиента проходит вертикально между верхней и нижней сторонами прямоугольника. При дальнейшем увеличении значения *fAngle* линия градиента продолжает вращаться по часовой стрелке, когда оно становится больше 90°. Но теперь первый цвет находится в правом верхнем, а второй — в левом нижнем углу прямоугольника.

Необязательный аргумент *bScale* указывает, подвержен ли угол поворота любому преобразованию, связанному с кистью.

О свойствах *WrapMode* и *Transform* объекта *LinearGradientBrush* я сказал выше. Среди четырех свойств, приведенных в таблице, находится прямоугольник, задаваемый конструктором явно (или неявно), и массив из двух цветов, используемых кистью:

Свойства *LinearGradientBrush* (выборочно)

Тип	Свойство	Доступ
RectangleF	*Rectangle*	Чтение
Color[]	*LinearColors*	Чтение/запись
WrapMode	*WrapMode*	Чтение/запись
Matrix	*Transform*	Чтение/запись

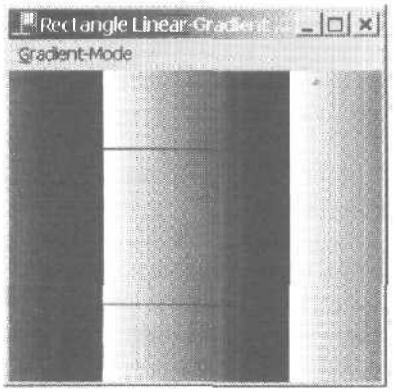
Кроме методов-двойников *TranslateTransform*, *ScaleTransform*, *RotateTransform*, *MultiplyTransform* и *ResetTransform* класса *TextureBrush*, в классе *LinearGradientBrush* есть еще два:

Методы *LinearGradientBrush* (выборочно)

```
void SetBlendTriangularShape(float fFocus)
void SetBlendTriangularShape(float fFocus, float fScale)
void SetSigmaBellShape(float fFocus)
void SetSigmaBellShape(float fFocus, float fScale)
```

Обычно градиент определяет одиночный переход от первого цвета ко второму. Эти два метода изменяют ход градиента так, что второй цвет переходит в первый, а затем обратно во второй. Оба аргумента (я опишу их ниже) могут меняться от 0 до 1,

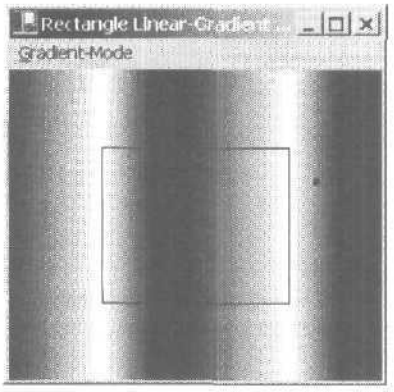
Рассмотрим пример. Вот исходный вид результата программы RectangleLinearGradientBrush, исполненной со значением свойства *LinearGradientMode*, заданным по умолчанию (*Horizontal*):



Градиент определяет переход от белого цвета (у левой стороны прямоугольника) к черному (у правой). Если сразу после оператора, создающего кисть, поместить оператор:

```
lgbrush.SetBlendTriangularShape(0.33f);
```

результат программы изменится;



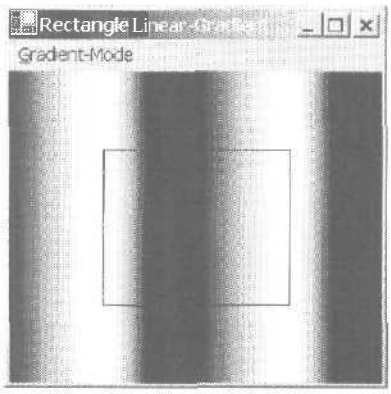
Новый градиент определяет переход от белого цвета (у левой стороны прямоугольника) к черному и обратно (у правой стороны прямоугольника). Аргумент

fFocus, равный 0,33, задает расположение максимумов черного цвета на 1/3 расстояния между сторонами прямоугольника.

Если заменить этот метод на:

```
lgbrush.SetSigmaBellShape(0.33f);
```

окно программы станет таким:



Полосы черного цвета становятся шире, а максимум интенсивности черного цвета перемещается в точку, расположенную на 1/3 расстояния между левой и Прагой сторонами прямоугольника,

В обоих методах аргумент *fScale* задет степень перехода первого цвета во второй. По умолчанию задано значение 1. Любое значение меньше 1 определяет неполный переход ко второму цвету. Если значение *JScale* равно 0, кисть будет образована лишь первым цветом, и градиента как такового не будет вообще,

Следующие свойства дополнительно расширяют возможности по управлению цветами градиента:

Свойства *LinearGradientBrush* (выборочно)

Тип	Свойство	Доступ
Blend	*Blend*	Чтение/запись
ColorBlend	*InterpolationColors*	Чтение/запись
bool	*GammaCorrection*	Чтение/запись

Blend и *ColorBlend* — довольно простые классы, у каждого из них всего лишь по паре изменяемых свойств, представляющих собой массивы одинакового размера. Два свойства *Blend* — это массивы значений *float* с именами *Positions* и *Factors*. Они задают коэффициенты (их значения варьируются от 0 до 1), используемые для определения относительных позиций двух цветов на линии градиента (соответственно позиция может варьироваться от 0 до 1). Два свойства *ColorBlend* — это массивы *Positions* и *Colors,* указывающие цвета, занимающие относительные позиции вдоль линии градиента.

Кисти с множественным градиентом

Последний тип кистей называется *PathGradientBrush*. В конструкторе соответствующего объекта определяется многоугольник (как вы уже знаете, просто массивом

точек), внутренняя область которого будет залита заданной кистью. Вместо прямоугольника можно просто задать объект *GraphicsPath*.

Конструкторы *PathGradientBrush*

```
PathGradientBrush(Point[] apt)
PathGradientBrush(PointF[] aptf)
PathGradientBrush(Point[] apt, WrapMode wm)
PathGradientBrush(PointF[] aptf, WrapMode wm)
PathGradientBrush(GraphicsPath path)
```

Давайте сразу познакомимся с этим видом кистей на практике: зададим в конструкторе *PathGradientBrush* треугольник и посмотрим, что произойдет:

TriangleGradientBrush.cs

```
//-------------------------------------------------
// TriangleGradientBrush.cs & 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TriangleGradientBrush: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new TriangleGradientBrush());
     }
     public TriangleGradientBrush()
     {
          Text = "Triangle Gradient Brush";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Point[] apt = { new Point(cx, 0),
                          new Point(cx, cy),
                          new Point(0, cy) };

          PathGradientBrush pgbrush * new PathGradientBrush(apt);

          grfx.FillRectangle(pgbrush, 0, 0, cx, cy);
     }
}
```

Программа особенно не изменилась: добавилось определение массива из трех точек (расположенных в правом верхнем, правом нижнем и левом нижнем углах отображаемой области), на основе этих точек затем создается объект *PathGradientBrush*. Однако результат впечатляет:



Явно, кто-то мудро определил параметры по умолчанию!

Заметьте: хотя я заливаю этой кистью весь клиентский прямоугольник, получается объект треугольной формы. Это результат определения кисти в виде треугольника. За пределами этого треугольника кисть прозрачна: заливка не влияет на то, что было в этой части клиентской области до вызова метода *FillRectangie*.

В объекте *PathGradientBrush* общего вида градиент формируется как между каждой парой точек, расположенной по сторонам заданного прямоугольника, так и между сторонами и центром многоугольника. Расположение центральной точки и цвета градиента определяют три свойства:

Свойства *PathGradientBrush* (выборочно)

Тип	Свойство	Доступ
PointF	*CenterPoint*	Чтение/запись
Color	*CenterColor*	Чтение/запись
Color[]	*SurroundColors*	Чтение/запись

Свойство *CenterPoint исходно* задано как среднее арифметическое координат вершин многоугольника, реально же эта точка может располагаться вне контуров многоугольника, если последний вогнутой формы. Свойство *CenterColor* первоначально установлено как *Color.Black*. Свойство *SurroundColors* инициализируется массивом из одного элемента со значением *Color.White*. Можно заносить в свойство *SurroundColors* любой массив, число элементов которого не превышает числа точек многоугольника, определяющего кисть,

Так, если прямо перед вызовом метода *FillRectangie* вставить в эту программу строку:

```
pgbrush.SurroundColors = new Color[] { Color.Red, Color.Green, Color.Blue };
```

точка (*cx*, 0) станет красной, точка (*cx*, *cy)* — зеленой, а точка (0, *cy)* — син::й. Попробуйте сделать это!

Вот программа, создающая многоугольник в форме пятиконечной звезды, белой в центре и черной по периферии.

StarGradientBrush.cs

```
//-------------------------------------------
// StarGradientBrush.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class StarGradientBrush: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new StarGradientBrush());
     }
     public StarGradientBrush()
     {
          Text = "Star Gradient Brush";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Point[] apt = new Point[5];

          for (int i = 0; i < apt.Length; i++)
          {
               double dAngle = (i * 0.8 - 0.5) * Math.PI;
               apt[i] = new Point(
                         (int)(cx * (0.50 * 0.48 * Math.Cos(dAngle))),
                         (int)(cy * (0.50 + 0.48 * Math.Sin(dAngle))));
          }
          PathGradientBrush pgbrush = new PathGradientBrush(apt);
          pgbrush.CenterColor = Color.White;
          pgbrush.SurroundColors = new Color[1] { Color.Black };
          grfx.FillRectangle(pgbrush, 0, 0, cx, cy);
     }
}
```



Во время исполнения программы даже видно, как GDI+ рисует градиенты, переходя от точки к точке многоугольника. Каждый следующий градиент рисуется поверх предыдущего. В результате создается впечатление, что звезда нарисована «без отрыва от бумаги»:

Объект *PathGradientBrush* можно определить и при помощи контура. Вот программа, в которой переопределен метод *DrawBall* из программы Bounce (см. главу И).

BouncingGradientBrushBall.cs

```
//-----------------------------------------------
// BouncingGradientBrushBall.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class BouncingGradientBrushBall: Bounce
{
     public new static void Main()
     {
          Application.Run(new BouncingGradientBrushBall());
     }
     public BouncingGradientBrushBall()
     {
          Text = "Bouncing Gradient Brush Ball";
     }
     protected override void DrawBall(Graphics grfx, Rectangle rect)
     {
          GraphicsPath path = new GraphicsPath();
          path.AddEllipse(rect);

          PathGradientBrush pgbrush = new PathGradientBrush(path);
          pgbrush.CenterPoint = new PointF((rect.Left + rect.Right) / 3,
                                           (rect.Top + rect.Bottom) / 3);
          pgbrush.CenterColor = Color.White;
          pgbrush.SurroundColors = new Color[] { Color.Red };

          grfx.FillRectangle(pgbrush, rect);
     }
}
```

Как вы, возможно, помните, метод *DrawBall* отвечает за рисование изображения мяча поверх битовой карты. Более ранняя версия этой программы просто рисовала красный эллипс при помощи объекта *Ellipse* с аргументом *Rectangle*. Эта версия определяет контур в виде аналогичного эллипса и создает на основе полученного контура объект *PathGradientBrush*. Обычно центр градиента располагается в центре эллипса, но определенный здесь метод сдвигает его немного влево и вверх. Для фигуры задан белый центральный цвет и красный периферийный. Исполнение метода завершается рисованием прямоугольника этой кистью (за пре-

делами границ эллипса она прозрачна). Полученное в результате изображение мяча выглядит реалистичнее, чем нарисованное прежней версией программы, так как белое пятно напоминает блик па настоящем мяче.



Мозаичное расположение кисти

Вот еще два полезных свойства объекта *PathGradientBrush*:

Свойства *PathGradientBrush* (выборочно)

Тип	Свойство	Доступ
RectangleF	*Rectangle*	Чтение
WrapMode	*WrapMode*	Чтение/запись

Rectangle — это неизменяемое свойство, значение которого рассчитывается при создании кисти. Оно определяет наименьший прямоугольник, в который может быть вписан многоугольник, определяющий кисть. Свойство *CenterPoint* не влияет на этот прямоугольник, т. е. задаваемая свойством *CenterPoint* точка вовсе не обязательно находится внутри этого прямоугольника.

Для кисти с множественным градиентом свойство *WrapMode* по умолчанию равно *WrapMode.Clamp*. Кроме установки свойства *WrapMode,* в конструкторе можно (но не обязательно) задавать значение, отличное от заданного по умолчанию. Я решил упомянуть свойства *Rectangle* и *WrapMode* вместе, так как эффект *WrapMode* сильно зависит от прямоугольника, определяемого свойством *Rectangle*.

Построим еще один треугольник, на этот раз фиксированного размера, занимающий левую верхнюю половину квадрата. Для установки способа расположения кисти, выбираемого при помощи меню, применяется версия конструктора с двумя аргументами.

TriangleTHe.cs

```
//-------------------------------------
// TriangleTile.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
```

см. след. стр.

```
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TriangleTile: PrintableForm
{
     const int iSide * SS;      // Сторона квадрата, определяющего треугольник
     MenuItem  miChecked;

     public new static void Main()
     {
          Application.flun(rtewTriangleTile());
     }
     public TriangleTileO ,
     {
          Text = "Triangle Tile";

          Menu = new MainMenu();
          Menu.MenuItems.Add("&Wrap-Mode");

          foreach (WrapMode wm in Enum.GetValues(typeof(WrapMode)))
          {
               MenuItem mi = new MenuItem();
               mi.Text     = wm.ToString();
               mi.Click   += new EventHandler(MenuWrapModeOnClick);
               Menu.MenuItems[0].MenuItems.Add(mi);
          }
          miChecked = Menu.MenuItems[0].MenuItems[0];
          miChecked,Checked *true;
     }
     void MenuWrapModeOnClick(object obj,  EventArgs ea)
     {
          miChecked.Checked - false;
          miChecked = (MenuItem) obj;
          miChecked.Checked = true;
          Invalidate();          ч
     }
     protected override void DoPage(Graphics grfx,  Color clr,  int cx,  int cy)
     {
          Point[] apt = { new Point(0,      0),
                          new  Point(iSide, 0),
                          new Point(0,      iSide)};

          PathGradientBrush pgbrush =
                    fiew PathGradientBrush(apt, (WrapMode) miChecked.Index);

          grfx.FillRectangle(pgbrush, 0, 0. cx, cy);
     }
}
```

Если у конструктора нет второго аргумента, *WrapMode*по умолчанию присваивается значение *WrapModeClamp*. Это значит, что многоугольник вообще не бу-

дет размножен. Если задано значение *WrapMode.Tile* (первоначальное значение свойства *WrapMode*), то многоугольник размножается по горизонтали и вертикали, заполняя всю заливаемую область (в данном случае это прямоугольник, ограничивающий клиентскую область программы).



Конечно, разные способы расположения кисти позволяют получать разнообразные эффекты. При использовании значения *WrapMode.TileFlipX* соседние многоугольники отражаются по вертикальной оси:



Аналогично значение *WrapMode.TileFlipY* определяет отражение соседних многоугольников по горизонтальной оси:



TwoTriangleTile.cs

```
//---------------------------------------------
// TwoTriangleTile.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

ciass TwoTriangleTile: PrintableForm
{
     const int iSide - 50;      // Сторона квадрата, определяющего треугольник

     public new static void Main()
     {
          Application.Run(new TwoTriangleTile());
     }
     public TwoTriangleTile()
     {
          Text = "Two-Triangle Tile";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
               // Определить треугольник и создать первую кисть.

          Point[] apt =
               {new Point(0, 0), new Point(iSide, 0), new Point(0, iSide)};

          PathGradient Brush pgbrush1 =
                         new PathGradientBrush(apt, WrapMode.TileFlipXY);

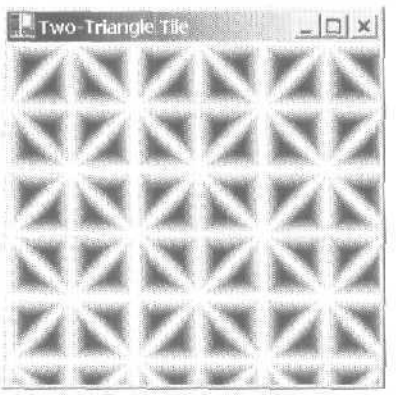
               // Определить другой треугольник и создать втору» кисть.

          apt * new Point[] {new Point(iSide, 0), new Point(iSide, iSide),
                             new Point(0, iSide)};

          PathGradientBrush pgbrush2 =
                         new PathGradientBrush(apt, WrapMode.TileFlipXY);

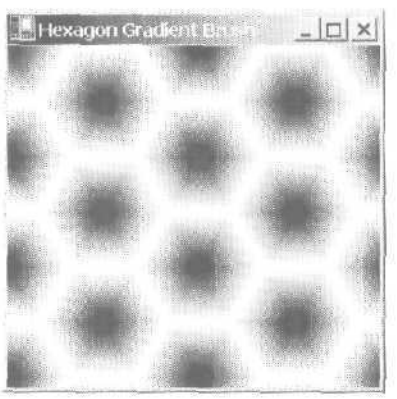
          grfx.FillRectangle(pgbrush1, 0, 0, ox, cy);
          grfx.FillRectangle(pgbrush2, 0, 0, cx, cy);
     }
}
```

Заметьте: второй многоугольник — это просто треугольник, занимающий правый **нижний** угол квадрата. Комбинация двух треугольных кистей позволяет залить клиентскую область программы целиком:



Поскольку в целом эффект выглядит как расположенные мозаикой квадраты, возникает вопрос: а нельзя ли нарисовать такой узор при помощи *PathGradient-Brush?* Ответ — нет, так как у объекта *PathGradientBrush* только один центр, а у каждого квадрата в этом узоре — по два центра (конечно, можно воспроизвести этот эффект, используя *TextureBrush*, поскольку эта кисть «рисует», размножая заданные изображения).

Экспериментировать с *PathGradientBrush* так забавно, что оторваться просто невозможно. Рассмотрим следующий орнамент в виде сот:



По черным лучам видно, что объект *PathGradientBrush* определяется шестиугольником, что, похоже, делает мозаичное расположение кисти невозможным. В вертикальных столбцах грани соседних шестиугольников соприкасаются. Но если размножить такой столбец по горизонтали, между фигурами определенно останутся промежутки. И в этом случае выход — в использовании пары кистей: одна рисует четные столбцы шестиугольников, другая — нечетные. Пара кистей образует мозаику шестиугольников, расположенных без промежутков как по горизонтали. так и по вертикали.

HexagonGradientBrush.cs

```
//-----------------------------------------------
// HexagonGradientBrush.cs © 2001 by Charles Petzold
//  ...  ........  ....  ....  ..  ..........
```

см. след. стр.

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HexagonGradientBrush: PrintableForm
{
     const float fSide = 50;  // Сторона шестиугольника (также равна его радиусу)

     public new static void Main()
     {
          Application.Run(new HexagonGradientBrush());
     }
     public HexagonGradientBrush()
     {
          Text = "Hexagon Gradient Brush";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
               // Вычислить полувысоту шестиугольника.

          float fHalf = fSide - (float) Math.Sin(Math.PI / 3);

               // Определить шестиугольник при помощи дополнительных вычислений.

          PointF[] aptf = {new PointF( fSide,         0),
                           new PointF( fSide * 1.5f,  0),
                           new PointF( fSide,         0),
                           new PointF( fSide / 2,     -fHalf),
                           new PointF(-fSide / 2,     -fHalf),
                           new PointF(-fSide,         0),
                           new PointF(-fSide * 1.5f,  0),
                           new PointF(-fSide,         0),
                           new PointF(-fSide / 2,     fHalf),
                           new PointF( fSide / 2,     fHalf) };

               // Создать первую кисть.

          PathGradientBrush pgbrush1 =
                              new PathGradientBrush(aptf, WrapMode.Tile);

               // Сместить шестиугольник и определить вторую кисть.

          for (int i = 0; i < aptf.Length; i++)
          {
               aptf[i].X += fSide * 1.5f;
               aptf[i].Y += fHalf;
          }
          PathGradientBrush pgbrush2 =
                              new PathGradientBrush(aptf, WrapMode.Tile);
```

см. след. стр.

```
            grfx.FillRectangle(pgbrush1, 0, 0, cx, cy);
            grfx.FillRectangle(pgbrush2, 0, 8, cx, cy);
        }
    }
```

Перья тоже могут быть кистями

Пока в этой главе обсуждались только кисти, хотя в названии упомянуты и перья. Дело в том, что из кистей можно создавать перья. Вот полный список конструкторов объекта *Pen*:

Конструкторы *Pen*

```
Pen (Color clr)
Pen (Color clr, float fWidth)
Pen (Brush brush)
Pen (Brush brush, float fWidth)
```

Создание объекта *Pen* из объекта *SolidBrush* эквивалентно созданию *Pen из* объекта *Color*, на основе которого создан объект *SolidBrush*.

В случае перьев весьма полезно помнить про вышеупомянутый эффект «взгляда через трафарет». Рисуя пером, созданным из кисти, вы в сущности создаете прорезь, сквозь которую видна кисть. Так, следующая программа создает объект *Pen* на основе *LinearGradientBrush*.

GradientPen.cs

```
//---------------------------------------
// GradientPen.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class GradientPen: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new GradientPen());
    }
    public GradientPen()
    {
        Text = "Gradient Pen";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush lgbrush = new LinearGradientBrush(
```

см. след, стр.

```
                                new Rectangle(0, 0, cx, cy),
                                Color.White, Color.Black,
                                LinearGradientMode.BackwardDiagonal);

            Pen pen = new Pen(lgbrush, Math.Min(cx, cy) / 25);

            pen.Alignment = PenAlignment.Inset;

            grfx.DrawRectangle(pen, 0, 0, cx, cy);
            grfx.DrawLine(pen, 0, 0, cx, cy);
            grfx.DrawLine(pen, 0, cy, cx, 0);
        }
f
```

Кисть определяется прямоугольником, заключающим в себе область рисования. Для свойства *LinearGradientMode* задано значение *BackwardDiagonal*. Это означает, что линия производного цвета проходит из левого верхнего угла прямоугольника в правый нижний. Если пером, созданным из этой кисти, нарисовать линию, совпадающую с линией производного цвета или параллельную ей, то она будет нарисована одним цветом, а не градиентом.



Хотя эта программа рисует все линии одним пером, вид градиента у разных линий разный. Цвет диагональной линии, проведенной из левого нижнего угла в правый верхний, переходит из черного в белый, а цвета горизонтальных и вертикальных линий — соответственно из черного в серый или из серого и белый.

В этой программе я использовал перо, минимальная толщина которого составляет 1/25 от ширины и высоты клиентской области. В следующей таблице показаны некоторые свойства класса *Pen,* связанные с толщиной пера:

Свойства Pen (выборочно)

Тип	Свойство	Доступ
float	*Width*	Чтение/запись
Matrix	*Transform*	Чтение/запись
PenAlignment	*Alignment*	Чтение/запись

Значение свойства *Width* выражается в глобальных координатах, но никогда не бывает меньше 1 пиксела. Если требуется перо толщиной в 1 пиксел, задайте толщину, равную 0.

Кроме свойства *Transform* класс *Pen* обладает обычным набором методов, задающих преобразования: *TranslateTransform*, *ScaleTransform*, *RotateTransform*, *MultiplyTransform* и *ResetTransform*. Однако преобразование *не влияет* ни на расположение и ориентацию нарисованных пером линий, ни на кисть, из которой создано перо. Преобразование влияет только на толщину пера. Для перьев лучше всего использовать такое преобразование масштабированием. Методом *ScaleTransform* (или задавая свойство *Transform* вручную) можно создавать перья с разной горизонтальной и вертикальной толщиной. Допустим, имеется перо толщиной 10 единиц. Если вызвать:

```
pen.ScaleTransform(2, 4);
```

толщина пера по горизонтали станет 20, а по вертикали — 40 единиц. Это значит, что толщина вертикальных линий, нарисованных этим пером, будет 20, а горизонтальных — 40 единиц, диагональные же будут иметь промежуточную толщину. Метод *RotateTransform* добавит к этому эффект вращения.

В программе GradientPen я устанавливал свойство *Alignment,* в которое можно занести одно из значений перечисления *PenAlignment*, определяемого в пространстве имен *System.Drawing.Drawing2D*:

Перечисление *PenAlignment*

Член	Значение
Center	0
Outset	1
Inset	2
Left	3
Right	4

Свойство *Alignment* управляет видом пера при рисовании прямоугольников и эллипсов. По умолчанию это свойство равно *PenAlignment.Center*. Это значит, что толстое перо будет центрировано относительно точки с заданными координатами. В программе GradientPen линии, нарисованные методом *DrawRectangle*, будут на половину своей толщины выходить за пределы клиентской области. Если в этой программе задать для пера свойство *PenAlignment.Inset*, то линии, нарисованные этим пером, не выйдут за пределы прямоугольника.

Следующие три свойства касаются объектов *Brush* и *Color,* используемых объектом *Pen*.

Свойства *Pen* (выборочно)

Тип	Свойство	Доступ
PenType	*PenType*	Чтение
Color	*Color*	Чтение/запись
Brush	*Brush*	Чтение/запись

Перечисление *PenType* определяется в пространстве имен *System.Drawing.Drawing2D*:

Перечисление *PenType*

Член	Значение
SolidColor	0
HatchFill	1
TextureFill	2
PathGradient	3
LinearGradient	4

Если перо создано с заданным цветом или на основе объекта *SolidBrush*, свойство *Color* объекта *Pen* является действительным, в противном случае действительно свойство *Brush*. Однако изменение свойства *Color* или *Brush* у существующего пера равносильно изменению типа этого пера.

Оформление пунктирных линий

Перья рисуют не только сплошные линии, но и линии, состоящие из точек и штрихов. Стиль такой линии задает свойство пера *DashStyle:*

Свойства *Pen* (выборочно)

Тип	Свойство	Доступ
DashStyle	*DashStyle*	Чтение/запись
float	*DashOffset*	Чтение/запись
float[]	*DashPattern*	Чтение/запись

Перечисление *DashStyle* определяется в пространстве имен *System.Drawing.Drawing2D*:

Перечисление *DashStyle*

Член	Значение
Solid	0
Dash	1
Dot	2
DashDot	3
DashDotDot	4
Custom	5

Вид точек и штрихов зависит от толщины пера и действующих преобразований, эту зависимость демонстрирует программа PenDashStyles.cs.

PenDashStyles.cs

```
//-----------------------------------------
// PenDashStyles.cs в 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
```

с/и. *след.* с *тр.*

```
class PenDashStyles: PrintableForm
{
     MenuItem miChecked;

     public new static void Main()
     {
          Application.Run(new PenDashStyles());
     }
     public PenDashStyles()
     {
          Text * "Pen Dash Styles";

          Menu = new MainMenu();
          Menu.MenuItems.Add("&Width");

          int[] aiWidth = { 1, 2, 5, 10, 15, 20, 25 };

          foreach (int iWidth 1л aiWidth)
               Menu.MenuItems[0].MenuItems.Add(iWidth.ToString(),
                                    new EventHandler(MenuWidthOnClick));

          miChecked = Menu.MenuItems[0].MenuItems[0];
          miChecked.Checked = true;
     }
     void MenuWidthOnClick(object obj, EventArgs ea)
     {
          miChecked.Checked = false;
          miChecked = (MenuItem) obj;
          miChecked.Checked = true;
          Invalidate();
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen pen = new Pen(clr);
          pen.Width = Convert.ToInt32(miChecked.Text);

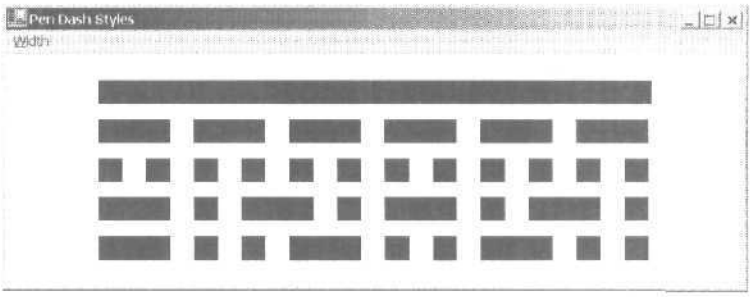
          for (int i = 0; i < 5; i++)
          {
               pen.DashStyle - (DashStyle) i;

               int y = (i + 1) * cy / 6;

               grfx.DrawLine(pen, cx / 8, y, 7 * ex / 8, y);

     }    *
}
```

Программа создает меню, позволяющее выбрать толщину пера — от 1 до 25. Затем она рисует в клиентской области пять равноудаленных друг от друга линий, оформленных при помощи первых пяти стилей. Вот какая морзянка получается, если выбрать перо толщиной 25:



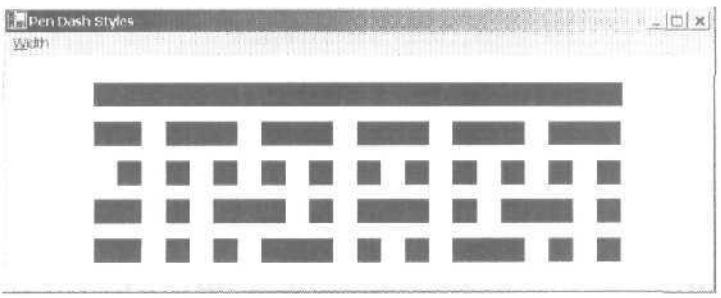
В следующем разделе я покажу, как управлять внешним видом больших штрихов и точек.

Чтобы нарисовать ломаную со штрих-пунктирным контуром, задействуйте метод *DrawLines* или контур. Не прибегайте к повторным вызовам *DrawLine*, так как при каждом вызове этого метода шаблон линии рисуется заново.

Свойство *DashOffset* позволяет управлять положением начальной точки шаблона пунктирной линии. Это свойство задает смещение точки начала шаблона, определяющего вид точек и штрихов данного стиля. Единица смещения — размер точки, величина смещения не зависит от толщины пера. Например, если вставить в PenDashStyles строку:

```
pen.DashOffset = 1;
```

шаблон линий сместится на размер одной точки влево и линии примут такой вид:



Тип значения свойства *DashOffset* — *float*, поэтому оно может быть дробным. Чтобы линии, оформленные стилями *DashDot* и *DashDotDot*, начинались не со штрихов, а с точек, присвойте *DashOffset* значение 4.

```
pen.DashOffset = 4;
```

Свойство *DashPattern* позволяет пользователю задавать свой шаблон штрих-пунктирной линии. Заданный этим свойством массив значений типа *float* указывает длину чередующихся штрихов и пробелов между ними. Единица всех размеров — размер точки. Вот пример:

```
float[] afDash = {2, 1, 4, 3};
pen.DashPattern = afDash;
```

DashStyle приобретает значение *Custom* после определения свойства *DashPattern*. Шаблон линии (нарисованной пером, заданным аргументом *pen*) состоит из штриха, длиной в 4 точки, пробела, еще одного штриха (длиной в 4 точки) и трех пробелов подряд.

Концы и соединения линий

В программе PenDashStyle при утолщении линии ее концы сохраняют исходную прямоугольную форму. Возможно, кому-то нравятся прямоугольные точки и черточки, но многие предпочитают придавать им закругленную форму.

Это сфера влияния свойств, задающих вид *концов* и *соединений* линий. Концы определяют вид линии там, где она начинается и заканчивается, а также вид точек и черточек. Соединение указывает, что происходит там, где встречаются две линии. Вот четыре основных свойства, определяющих форму концов и соединений ЛИНИЙ:

Свойства *Pen* (выборочно)

Тип	Свойство	Доступ
LineCap	*StartCap*	Чтение/запись
LineCap	*EndCap*	Чтение/запись
DashCap	*DashCap*	Чтение/запись
LineJoin	*Linejoin*	Чтение/запись

Хотелось бы начать с *Linejoin*, поскольку оно самое простое из этих свойств. Его значения составляют перечисление *LineJoin*:

Перечисление *Linejoin*

Член	Значение	Описание
Miter	0	Остроконечное соединение (задано по умолчанию)
Bevel	1	Соединение со срезанной вершиной
Round	2	Закругленное соединение
MiterClipped	3	Остроконечное соединение с ограничением

Свойство *Linejoin* влияет только на соединяемые линии, т. е. на составные линии, нарисованные методом *DrawLines*, или на соединяемые линии контура. Следующая программа рисует простую V-образную ломаную и применяет к ней различные значения *Linejoin*.

LineJoins.cs

```
//-----------------------------------------
// LineJoins.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class LineJoins: PrintableForm
{
    public new static void Main()
    {
```

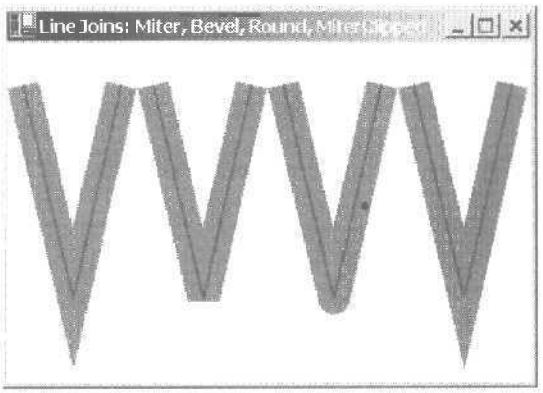
см. след. стр.

```
        Application.Run(new LineJoins());
    }
    public LineJoins()
    {
        Text в "Line Joins: Miter, Bevel, Round, MiterClipped";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int ey)
    {
        Pen      penNarrow = new Pen(clr);
        Pen      penWide   = new Pen(Color.Gray, cx / 16);
        Point[] apt       * { new Point(1 * cx / 32, 1 * cy / 8),
                              new Point(4 * cx / 32, 6 * cy / 8),
                             ' new Point(7 * cx / 32, 1 * ey / 8) K    ;

        for (int i = 0; i < 4; i++)
        {
            penWide.LineJoin = (LineJoin) i;

            grfx.DrawLines(penWide, apt);
            grfx.DrawLines(penNarrow,  apt);
            grfx.TranslateTransform(cx / 4, 0);
        }
    }
}
```

А вот как выглядит ее результат:



Программа устанавливает свойство *LineJoin* для толстого пера серого цв;:та. Тонкая черная линия показывает ход реальной геометрической линии. Нетрудно заметить, что соединение, заданное значением *MiterClipped*,по виду почти не отличается от соединения, определяемого *Miter*. Но чем ближе концы V-образной фигуры, тем тоньше и острее будет вершина соединения. Вершина соединения *MiterClipped* в аналогичной ситуации обрезается, результат напоминает эффект значения *Bevel*. Есть причина, вынуждающая ограничивать длину вершин остроконечных соединений. Дело в том, что с уменьшением угла между соединяемыми линиями вершина соединения может стать слишком длинной, Например, длина

соединения двух линий толщиной 1 дюйм, которые сходятся под углом 1°, равна 135 см![1] У класса *Pen* есть специальное свойство, ограничивающее длину вершины соединений. Оно действует, когда значение *LineJoin* равно *MiterClipped*:

Свойства *Pen* (выборочно)

Тип	Свойство	Доступ
float	*MiterLimit*	Чтение/запись

MiterLimit обрезает вершину остроконечного соединения на расстоянии, равном *penMiterLimit* x *penWidth / 2*. По умолчанию оно равно 10. Если свойство *Width* пера равно 20, вершина соединения окажется на расстоянии не более 100 единиц от точки пересечения теоретических линий.

А теперь рассмотрим свойство *DashCap*, определяющее вид точек и штрихов штрих-пунктирных линий. У него может быть одно из значений следующего перечисления:

Перечисление *DashCap*

Член	Значение
Flat	0
Round	2
Triangle	3

Вот вариация программы PenDashStyles, которая рисует линию, оформленную стилем *DashDotDot*, с использованием трех различных значений *DasbCap*.

PenDashCaps.cs

```
//-----------------------------------------
// PenDashCaps.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class PenDashCaps: PrintableForm
{
     MenuItem miChecked;

     public new static void Main()
     {
          Application.Run(new PenDashCaps());
     }
     public PenDashCaps()
```

см. след. стр.

[1] Если толщина линии равна w, а угол между соединяемыми линиями — α, нетрудно показать, что вершина соединения будет расположена на расстоянии $(w/2)/\sin(\alpha/2)$ от реальной точки пересечения линий.

```
    {
        Text = "Pen Dash Gaps: Flat, Round, Triangle";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Width");

        int[] aiWidth = { 1, 2, 5, 10, 15, 20, 25 };

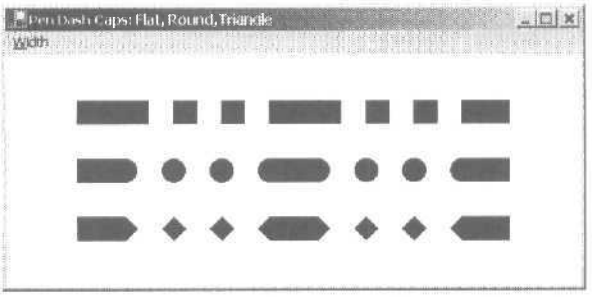
        foreach (int iWidth in aiWidth)
            Menu.MenuItems[0].MenuItems.Add(iWidth.ToString(),
                                    new EventHandler(MenuWidthOnClick));

        miChecked = Menu.MenuItems[0].MenuItems[0];
        miChecked.Checked = true;
    }
    void MenuWidthOnClick(object obj, EventArgs ea)
    {
        miChecked.Checked = false;
        miChecked = (MenuItem) obj;
        miChecked.Checked = true;
        Invalidate();
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        pen pen = new Pen(clr, Convert.ToInt32(miChecked.Text));
        pen.DashStyle = DashStyle.DashDotDot;

        foreach (DashCap dc in Enum.GetValues(typeof(DashCap)))
        {
            pen.DashCap = dc;

            grfx.DrawLine(pen, cx / 8, cy / 4, 7 * cx / 8, cy / 4);
            grfx.TranslateTransform(0, cy / 4);
        }
    }
}
```

Вот что рисует эта программа, если выбрать толщину пера 25:

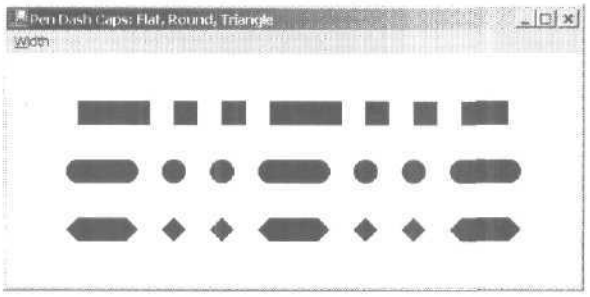


Это не совсем то, что надо, так как в начале и конце эти линии все равно сохраняют квадратную форму. Внешний вид концов линии определяется свойствами

StartCap и *EndCap*, оба этих свойства имеют тип *LineCap*. Чтобы согласовать вид концов линий со значением свойства *DashCap*, можно вставить в программу PenDashCaps оператор:

```
pen.StartCap = pen.EndCap = (LineCap) (int) dc;
```

Результат программы выглядит так:



Но линии с закругленными и треугольными концами не вполне выровнены с линией, у которой прямоугольные концы. Как я покажу ниже, причина этого в том, что закругленные и треугольные концы выходят за геометрические концы линии, Однако полная ширина точек и штрихов выдерживается независимо от стиля концов линии,

Вот полный список значений перечисления *LineCap:*

Перечисление *LineCap*

Член	Значение
Flat	0x00
Square	0x01
Round	0x02
Triangle	0x03
NoAnchor	0x10
SquareAnchor	0x11
RoundAnchor	0x12
DiamondAnchor	0x13
ArrowAnchor	0x14
AnchorMask	0xF0
Custom	0xFF

А вот программа, рисующая толстые линии с использованием всех значений этого перечисления. Толщина линии зафиксирована значением свойства *FontHeight*. В дополнение программа рисует тонкие линии, показывающие геометрические концы каждой из линий.

LineCaps.cs

```
//-------------------------------------
// LineCaps.cs © 2001 by Charles Petzold
//-------------------------------------
```

см. след. стр.

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class LineCaps: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new LineCaps());
     }
     public  LineCaps()
     {
          Text = "Line Caps";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen   penWide   = new Pen(Color.Gray, Font.Height);
          Pen   penNarrow = new Pen(clr);
          Brush brush     = new SolidBrush(clr);

          foreach (LineCap lc in Enum.GetValues(typeof(LineCap)))
          {
               grfx.DrawString(lc.ToString(), Font, brush,
                               Font.Height, Font.Height / 2);

               penWide.StartCap = lc;
               penWide.EndCap   = lc;

               grfx.DrawLine(penWide, 2 * cx / 4, Font.Height,
                                      3 * cx / 4, Font.Height);

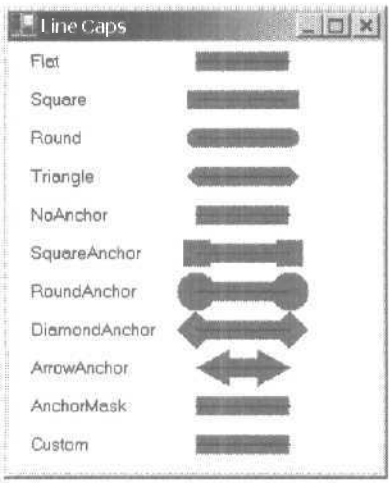
               grfx.DrawLine(penNarrow, 2 * cx / 4, Font.Height,
                                        3 * cx / 4, Font.Height);

               grfx.TranslateTransform(0, 2 * Font.Height);
          }
     }
}
```

Результат этой программы см. на обороте.

Имейте в виду, что я применяю одни и те же значения перечисления для оформления обоих концов линии. При желании вы можете использовать разные значения.

Значение *NoAnchor* дает тот же результат, что и *Flat*. Эффект значений *SquareAnchor*, *RoundAnchor* и *DiamondAncbor* аналогичен *Square, Round* и *Triangle* соответственно (как можно предположить по именам значений перечисления) за исключением того, что в этом случае концы линий несколько больше.



Если вам мало разнообразия форм концов линий, предоставляемого перечислением *LineCap*, можно задать для свойств пера *StartCap* и/или *EndCap* значение *LineCap.Custom*, и использовать следующие свойства:

Свойства *Pen* (выборочно)

Тип	Свойство	Доступ
CustomLineCap	*CustomStartCap*	Чтение/запись
CustomLineCap	*CustomEndCap*	Чтение/запись

Класс *CustomLineCap* (определяемый в пространстве имен *System.Drawing.Drawing2D*) позволяет задавать пользовательские контуры концов линии. Кроме того, класс *AdjustableArrowCap*, производный от *CustomLineCap*, позволяет рисовать концы линий в виде стрелок и предоставляет более широкие возможности по управлению размером стрелок и заполнением их контуров.

ГЛАВА

18



Текстовые поля, списки и управляющие стрелки

Почти в каждой программе для Windows пользователю приходится вводить текст — так было и будет. В главе 6 я рассказывал, как устанавливать обработчики событий *KeyDown*, *KeyUp* и *KeyPress* для получения и отображения на экране данных, вводимых с клавиатуры. Однако во многих простых случаях можно задействовать элемент управления, который обычно называют *полем редактирования* (edit control), а в .NET Framework — *текстовым полем* (text box).

На практике используются различные текстовые поля: от небольшого однострочного поля ввода до многострочного элемента управления с автоматическим переносом строк, как в программе Notepad (Блокнот). Написание клона Notepad — типичное упражнение для таких книг, как эта, и в данной главе мы выполним большую часть этого упражнения. (В главе 21 я добавлю в программу поддержку печати, а в главе 24 — поддержку буксировки.)

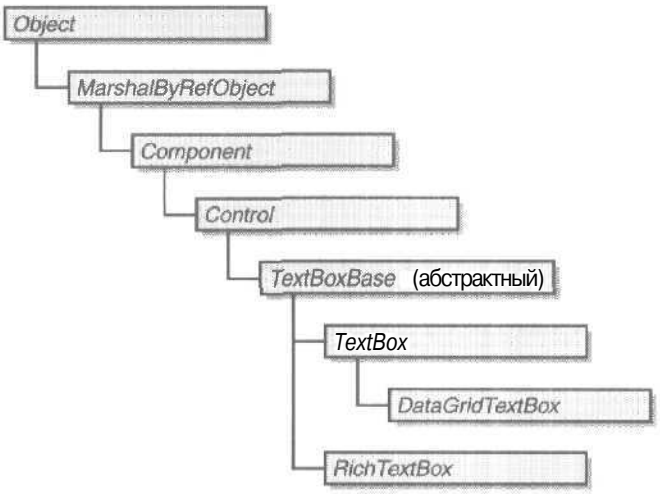
Кроме того, здесь я рассмотрю и другие важные типы элементов управления: *окно списка* (list box) — список элементов с прокруткой, позволяющий выбирать элементы, и *комбинированное окно* (combo box), сочетающее текстовое поле и окно списка. При работе с числами может потребоваться и *управляющие стрелки* (up-down control), который часто называют *счетчиком с прокруткой* (spin box).

Однострочные текстовые поля

Простейшее текстовое поле называется *TextBox*. Как видно из показанной ниже иерархии классов, *TextBox* наследует абстрактному классу *TextBoxBase*.

Еще одним наследником *TextBoxBase* является *RichTextBox*, Элемент управления *RichTextBox* обладает большей функциональностью по сравнению с *TextBox* аналогично тому, как Microsoft WordPad обладает большей функциональностью по

сравнению с Notepad. В *TextBox* (как и в Notepad) можно указать шрифт, но шрифт применяется ко всему документу. В *RichTextBox* (как и в WordPad) можно определить различные шрифты и прочее форматирование для различных частей документа.



Самое важное свойство текстовых полей — *Text:* именно оно содержит текст, вводимый в поле пользователем. Кроме того, можно инициализировать текст, показываемый в поле, и ограничивать длину текста:

Свойства *TextBoxBase* (выборочно)

Тип	Свойство	Доступ
string	*Text*	Чтение/запись
int	*MaxLength*	Чтение/запись
int	*TextLength*	Чтение
string]	*Lines*	Чтение/запись

Свойство *TextLength* — это то же самое, что и *TextLength.*

Свойство *Lines* можно было бы назвать *Paragraphs* (абзацы), так как оно разбивает документ на текстовые блоки, завершающиеся символами конца строки. Эти блоки показываются в виде строк текста, если перенос строк отключен, и в виде абзацев, если включен.

Зачастую в диалоговых окнах достаточно получить находящийся в поле текст в момент нажатия пользователем кнопки OK. Но в некоторых диалоговых окнах нужно постоянно отслеживать ввод пользователя. Так, диалоговое окно может разрешать нажимать кнопку OK, только когда текстовое поле содержит какую-то допустимую информацию. Самое полезное событие текстового поля на самом деле определено в *Control:*

События *Control* (выборочно)

Событие	Метод	Делегат	Параметр
TextChanged	*OnTextChanged*	*EventHandler*	*EventArgs*

Следующая программа создает текстовое поле и надпись. Устанавливая обработчик события *TextChanged* текстового поля, она следит за вводом текста в ппле и дублирует этот текст в надписи.

TextBoxDemo.cs

```
//---------------------------------------
// TextBoxDemo.cs © 2001 by Charles Petzold
//_____-·_________---_______--__'__·----------
using System;
using System.Drawing;
using System.Windows.Forms;

ciass TextBoxDemo: Form
{
     Label label;

     public static void Main()
     {
          Application.Run(new TextBoxDemo());
     }
     public TextBoxDemo()
     {
          Text = "TextBox Demo";

               // Создаем текстовое поле.

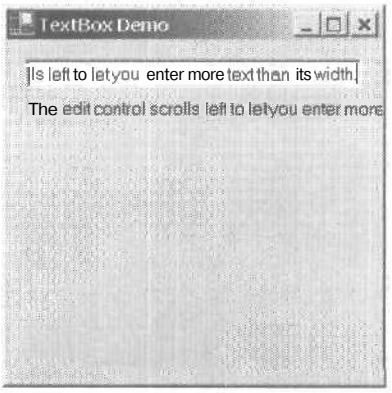
          TextBox txtbox     = new TextBox();
          txtbox.Parent      - this;
          txtbox.Location    - new Point (Font.Height, Font.Height);
          txtbox.Size        = new Size(ClientSize.Width - 2 * Font.Height,
                                        Font.Height);
          txtbox.Anchor      |= AnchorStyles.Right;
          txtbox.TextChanged +=new EventHandler(TextBoxOnTextChanged);

               // Создаем надпись.

          label          = new Label();
          label.Parent   = this;
          label.Location =new  Point(Font.Height,3 *  Font.Height);
        • label.AutoSize * true;
     }
     void TextBoxOnTextChanged(object obj, EventArgs ea)
     {
          TextBox txtbox = (TextBox) obi;

          label.Text - txtbox.Text;
     }
}
```

Так как я присвоил свойству *Anchor* текстового поля значение, при увеличении ширины формы поле будет растягиваться. Но текущий размер текстового поля не ограничивает количество вводимого текста. При достижении вводимым текстом правой границы поля текст будет автоматически прокручиваться влево. В этом случае возможно, что надпись не сможет отобразить весь введенный текст, даже несмотря на то, что ее свойству *AutoSize* присвоено значение *true*:



По умолчанию цветами фона и шрифта текстового поля являются *SystemColors.Window* и *SystemColors.WindowText* в отличие от *SystemColors.Control* и *SystemColors.ControlText*, используемых формой. По умолчанию свойство *Font* наследуется текстовым полем от предка. Свойство *BorderStyle* по умолчанию имеет значение *BorderStyle.Fixed3D*, что придает полю «утопленный» вид. Можно также присвоить этому свойству значение *None* или *FixedSingle*.

Я хотел бы, чтобы вы поэкспериментировали с программой TextBoxDemo, или с Notepad, или с любым другим текстовым полем любого приложения Windows. Как вам, несомненно, известно, каретка (caret) в текстовом поле указывает точку вставки — место, где появится вводимый текст. Каретку можно передвигать куда угодно в пределах всего блока текста клавишами управления курсором. Кроме того, можно изменять положение каретки щелчком мыши.

Держа клавишу Shift нажатой и перемещая курсор клавишами-стрелками, вы определите *выделение* (selection) — блок текста, выделенный в текстовом поле обратным цветом. Можно также определять выделение, используя мышь, перемещая ее с нажатой кнопкой.

Важный момент: если в поле выделен текст, каретка всегда находится в начале или в конце выделения. Это важно для понимания того, что свойства, в которых хранится информация о выделении, предоставляют сведения и о положении каретки. Всю эту информацию содержат четыре свойства *TextBoxBase*:

Свойства *TextBoxBase* (выборочно)

Тип	Свойство	Доступ
int	*SelectionStart*	Чтение/запись
int	*SelectionLength*	Чтение/запись
string	*SelectedText*	Чтение/запись
bool	*HideSelection*	Чтение/запись

Свойство *SelectionStart* — это индекс с нулевой базой, показывающий позицию символа, с которого начинается выделение. Если *SelectionStart* равно 0, выделение начинается с самого начала текстового поля.

Свойство *SelectionLength* показывает количество символов в выделении. Если оно равно 0, никакого текста не выделено, а *SelectionStart* показывает позицию каретки. Если и *SelectionStart,* и *SelectionLengtb* равны 0, каретка находится в самом начале текстового поля.

Если *SelectionLengtb* больше 0, точной информации о положении каретки нет. В зависимости от того, как выделялся текст (при помощи клавиш «стрелка-влево» или «стрелка-вправо» или перемещением мыши в определенном направлении), каретка может находиться в начале или в конце выделения. Позицией каретки будет или *SelectionStart,* или *(SelectionStart + SelectionLengtb).*

Если ничего не выделено, *SelectedText* ~ это пустая строка. В противном случае это строка, содержащая выделенный текст. На самом деле *SelectionLengtb* — просто сокращенная запись *SelectedText.Length.*

Заметьте: свойство *SelectedText* доступно для записи. Если программа дол» на удалить выделенный текст из поля (оставив при этом невыделенный текст), нужно просто присвоить ему пустую строку:

```
txtbox.SelectedText ="";
```

Вставить текст в месте, где находится каретка, позволяет оператор:

```
txtbox.SelectedText = "insert text";
```

Если при выполнении этого оператора в поле выделен какой-то текст, то вставляемый текст заменит выделение. Если ничего не выделено, текст вставится в позиции каретки.

Свойство *HideSelection* обычно имеет значение *true.* Это значит, что, когда текстовое поле теряет фокус ввода, выделение перестает показываться другим цветом. Это стандартное поведение. Далее в этой главе я буду использовать значение *HideSelection,* отличное от значения по умолчанию, в немодальном диалоговом окне поиска и замены.

У элемента управления *TextBox* нет события, позволяющего программе определить, что выделение изменилось. Такое событие имеется у элемента управления *RichTextBox* — оно называется *SelectionCbanged.*

Методы *Select* и *SelectAll* позволяют определять выделение в текстовом поле:

Методы TextBoxBase (выборочно)

```
void Select(int iStart, int iLength)
void SelectAll()
void Clear()
```

Метод *Select* равносилен присваиванию значений свойствам *SelectionStart* и *SelectionLength,* метод *SelectAll* — присваиванию *SelectionStart* значения 0, а *SelectionLengtb* — значения *TextLength.* Метод *Clear* равносилен присваиванию свойству *Text* пустой строки.

Экспериментируя с TextBoxDemo, вы могли заметить, что текстовое поле автоматически поддерживает работу с буфером обмена. Можно нажать Ctrl+X, чтобы удалить выделенный текст и копировать его в буфер обмена, Ctrl+C — чтобы копировать текст в буфер обмена без удаления, и Ctrl+V — чтобы вставить текст из буфера обмена. Ниже вы узнаете, как выполнять эти операции программно. Кроме того, стандартное текстовое поле обладает возможностью отмены действий (undo). После выделения текста и его удаления клавишей Delete, можно вернуть этот текст обратно, нажав Ctrl+Z. Это только одноуровневая отмена действий: при повторном нажатии Ctrl+Z удаленный текст снова исчезает, еще одно нажатие восстанавливает текст.

Многострочные текстовые поля

Превратить однострочное текстовое поле в многострочное довольно легко - нужно лишь задать *true* свойству *Multiline* и (вполне возможно, что это тоже нужно) сделать текстовое поле достаточно большого для показа нескольких строк размера. Однако для работы с многострочными текстовыми полями используется несколько дополнительных свойств.

Вот *Multiline* и еще пара свойств, относящихся к многострочным полям:

Свойства *TextBox* (выборочно)

Тип	Свойство	Доступ	Значение по умолчанию
bool	*Multiline*	Чтение/запись	*false*
boo!	*WordWrap*	Чтение/запись	*true*
ScrollBars	*ScrollBars*	Чтение/запись	*ScrollBarsNone*

Свойства *Multiline* и *Wordwrap* определены в *TextBoxBase* и наследуются также и *RichTextBox*. Свойство *ScrollBars,* определенное в *TextBox*, принимает одно из этих значений:

Перечисление *ScrollBars*

Член	Значение
None	0
Horizontal	1
Vertical	2
Both	3

При присваивании свойству значений *Horizontal*, *Vertical* или *Both* полосы прокрутки появляются, даже если они не нужны, т. е. когда весть текст умещается в поле и прокрутка не требуется. Если полосы прокрутки в данный момент не нужны, они отключаются. Независимо от значения свойства *ScrollBars* горизонтальная полоса прокрутки не показывается, если *Wordwrap* имеет значение *true.*

У *RichTextBox* также есть свойство *ScrollBars,* но другого типа — *RichTextBoxScrollBars* — и оно позволяет четче определить, когда должны появляться полосы прокрутки:

Перечисление RichTextBoxScrollBars

Член	Значение
None	0
Horizontal	1
Vertical	2
Both	3
ForcedHorizontal	['
ForcedVertical	18
ForcedBoth	19

Если используются члены перечисления *Horizontal, Vertical* и *Both,* полосы прокрутки показываются только в случае необходимости. Если применяются члены перечисления, начинающиеся со слова *Forced,* полосы прокрутки показываются в любом случае.

В многострочных текстовых полях нельзя центрировать текст по вертикали. Выравниванием текста управляет в основном свойство *TextAlign*:

Свойства *TextBox* (выборочно)

Тип	Свойство	Доступ
HorizontalAlignment	*TextAlign*	Чтение/запись

HorizontalAlignment является перечислением:

Перечисление *HorizontalAlignment*

Член	Значение
Left	0
Right	1
Center	2

Как вам известно, клавиши Tab и Enter в диалоговых окнах или в любой форме с дочерними элементами управления имеют специальное значение. Tab служит для перемещения фокуса между элементами управления; при нажатии Enter обычно делается то же, что и при щелчке кнопки OK. Для однострочных текстовых полей, используемых в диалоговых окнах, вам, вероятно, будет нужно, чтобы клавиши Tab и Enter работали как обычно. Однако в случае многострочных текстовых полей вам скорее всего потребуется, чтобы Tab и Enter обрабатывало само текстовое поле. Тогда следующим свойствам присвойте *true*:

Свойства *TextBox* (выборочно)

Тип	Свойство	Доступ	Значение по умолчанию
boo/	*AcceptsReturn*	Чтение/запись	*true*
bool	*AcceptsTab*	Чтение/запись	*false*

Свойство *AcceptsReturn* определено в *TextBox;* а *AcceptsTab* — в *TextBoxBase.* Если свойству *AcceptButton* родительской формы не присвоено значение, текстовое поле будет обрабатывать нажатие Enter независимо от значения *AcceptsReturn.*

Клонирование Notepad

Теперь мы готовы приступить к разработке клона Notepad — программы Notepad Clone. Чтобы представить код в виде управляемых частей, я буду наследовать каждую последующую версию программы от предыдущей. Получится несколько уровней наследования.

Первый модуль создает текстовое поле и использует *DockStyle.Fill*, чтобы оно заполнило клиентскую область формы. Конструктор содержит присваивание значений нескольким свойствам, относящимся к многострочным текстовым полям.

NotepadCloneNoMenu.cs

```
//--------------------------------------------------------
// NotepadCloneNoMenu.cs © 2001 by Charles Petzold
//--------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class NotepadCloneNoMenu: Form
{
     protected TextBox txtbox;

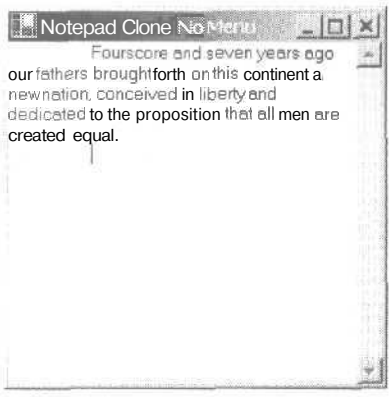
     public static void Main()
     {
          Application.Run(new NotepadCloneNoMenu());
     }
     public NotepadCloneNoMenu()
     {
          Text= "Notepad Clone No     Menu";

          txtbox             * new TextBox();
          txtbox.Parent      * this;
          txtbox.Dock        = DockStyle.Fill;
          txtbox.BorderStyle = BorderStyle.None;
          txtbox.Multiline   * true;
          txtbox.ScrollBars  - ScrollBars.Both;
          txtbox.AcceptsTab  = true;
     }
}
```

Эта небольшая программа обладает достаточной функциональностью; ее работа во многом напоминает работу Notepad, когда включен пункт меню Word Wrap (Перенос по словам). Ниже показана программа NotepadCloneNoMenu с введенным в нее текстом,

Прежде чем приступить к созданию меню в программе Notepad Clone, я хотел бы написать код, обращающийся к реестру Windows. Как вы, наверное, знаете, при изменении в Notepad состояния пункта меню Word Wrap (Перенос по словам) или шрифта при следующем запуске Notepad используются новые значения параметров. Можно посмотреть все параметры, которые Notepad хранит в реестре, запу-

стив редактор реестра Regedit.exe и открыв раздел реестра HKEY_CURRENT_USER\Software\Microsoft\Notepad.



Кроме запоминания переноса по словам и шрифта, Notepad хранит в реестре размер окна и позицию окна на рабочем столе. При перемещении или изменении размера окна Notepad, завершении работы программы и ее повторном запуске окно программы появится на том же месте и будет иметь тот же размер.

Может показаться, что возможность сохранения положения и размера реализуется так: при завершении работы программы свойство *DesktopBounds* формы сохраняется в реестре, а при повторном запуске программы этому свойству присваиваются значения из реестра. Эта схема действительно работала бы прекрасно, если б не *свертывание* (minimize) и *развертывание* (maximize).

Давайте рассмотрим эти возможности. Когда вы развертываете любую Windows-программу и затем восстанавливаете ее, положение и размер окна становятся такими же, как до развертывания. Аналогично, когда вы свертываете и затем восстанавливаете программу, программа также возвращается к прежним положению и размеру. Значит, Windows сохраняет положение и размер окна программы перед свертыванием/развертыванием. Программа, использующая Win32 API, обращается к этой информации через структуру *WINDOWPLACEMENT*, где такое положение называется *нормальным* (normal position). Структура *WINDOWPLACEMENT* применяется в функциях API *GetWindowPlacement* и *SetWindowPlacement*.

Программа Windows Forms не всегда имеет прямой доступ к нормальному положению. При развертывании формы свойство *DesktopBounds* отражает *развернутые* (maximized) положение и размер. Если у формы есть рамка, координаты положения развернутой формы становятся отрицательными, так как рамка в таком положении оказывается за пределами рабочего стола. Когда форма свернута, свойства *X* и *Y* прямоугольника *DesktopBounds* имеют специальные значения, равные -32000, свойства *Height* и *Width* являются размерами кнопки свернутой программы на панели задач Windows. И только когда форма не является свернутой или развернутой, *DesktopBounds* отражает нормальное положение формы.

Вы уже поняли суть проблемы? Проблема проявляется, когда программа завершается в свернутом или развернутом состоянии. Поскольку *DesktopBounds* не отражает нормальное положение формы, *DesktopBounds* в этом случае — не то, что программа должна сохранить в реестре.

По этой причине в программе Windows Forms, запоминающей в реестре свое положение в нормальном состоянии, надо выделить поле для хранения этого положения. (В программе, которую вы вскоре увидите, я назвал это поле *rectNormal.)* Этому полю нужно присваивать значение *DesktopBounds* в конструкторе и затем заново присваивать значение *DesktopBounds* при вызове *OnMove* или *OnResize,* но только когда форма не свертывается/развертывается. Проверить, свернуто окно или развернуто, позволяет свойство *WindowState:*

Свойства *Form* (выборочно)

Тип	Свойство	Доступ
FormWindowState	*WindowState*	Чтение/запись

В перечисление *FormWindowState* входят следующие члены:

Перечисление *FormWindowState*

Член	Значение
Normal	0
Minimized	1
Maximized	2

Очередная версия Notepad Clone унаследована от класса NotepadCloneNoMenu. В ней реализована работа с реестром, который служит для сохранения и восстановления положения и размера окна. Как я упоминал, полю *rectNormal* присваивается значение в конструкторе (когда свойство *WindowState* всегда имеет значение *FormWindowStateNormal*) и в методах *OnMove* и *OnResize,* если *WindowState* равно *FormWindowStateNormal.*

NotepadCloneWithRegistry.cs

```
//-----------------------------------------------------
// NotepadCloneWithRegistry.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Windows.Forms;

class NotepadCloneWithRegistry: NotepadCloneNoMenu
{
     Rectangle        rectNormal;
     protected string strProgName;
     string           strRegKey     *
                                "Software\\ProgrammingWindowsWithCSharp\\";
     const string     strWinState  = "WindowState";
     const string     strLocationX = "LocationX";
     const string     strLocationY = "LocationY";
     const string     strWidth     * "Width";
     const string     strHeight    - "Height";
```

см. след. стр.

```
public new static void Main()
{
    Application.Run(new NotepadCloneWithRegistry());
}
public NotepadCloneWithRegistry()
{
    Text = StrProgName = "Notepad Clone with Registry";
    rectNormal = DesktopBounds;
}
protected override void OnMove(EventArgs ea)
{
    base.OnMove(ea);

    if (WindowState == FormWindowState.Normal)
        rectNormal = DesktopBounds;
}
protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);

    if (WindowState == FormWindowState.Normal)
        rectNormal = DesktopBounds;
}
protected override void OnLoad(EventArgs ea)
{
    base.OnLoad(ea);

        /У Формируем полный раздел реестра.

    strRegKey - strRegKey + strProgName;

        // Загружаем информацию из реестра.

    RegistryKey regkey = Registry.CurrentUser.OpenSubKey(strRegKey);

    if (regkey != null)
    {
        LoadRegistryInfo(regkey);
        regkey.Close();
    }
}
protected override void OnClosed(EventArgs ea)
{
    base.OnClosed(ea);

        // Сохраняем информацию s реестре.

    RegistryKey regkey -
                Registry.CurrentUser.OpenSubKey(strRegKey, true);
```

см. след. стр.

```
            if (regkey == null)
                regkey = Registry.CurrentUser.CreateSubKey(strRegKey);

            SaveRegistryInfo(regkey);
            regkey.Close();
        }
        protected virtual void SaveRegistryInfo(RegistryKey regkey)
        {
            regkey.SetValue(strWinState,   (int) WindowState);
            regkey.SetValue(strLocationX, rectNormal.X);
            regkey.SetValue(strLocationY, rectNormal.Y);
            regkey.SetValue(strWidth,     rectNormal.Width);
            regkey.SetValue(strHeight,    rectNormal.Height);
        }
        protected virtual void LoadRegistryInfo(RegIstryKey regkey)
        {
            int x  - (int) regkey.GetValue(strLocationX, 100);
            int y  = (int) regkey.GetValue(strLocationY, 100);
            int cx = (int) regkey.GetValue(strWidth, 380);
            int cy = (int) regkey.GetValue(strHeight, 300);

            rectNormal = new Rectangle(x, y, cx, cy);

                // приводим прямоугольник в соответствие с размерами экрана
                //(размеры экрана могли измениться).

            Rectangle rectDesk = SystemInformation.WorkingArea;

            rectNormal.Width  = Math.Min(rectNormal.Width,  rectDesk.Width);
            rectNormal.Height - Math.Min(rectNormal.Height, rectDesk.Height);
            rectNormal.X -= Math.Max(rectNormal.Right  - rectDesk.Right,  0);
            rectNormal.Y -= Math.Max(rectNormal.Bottom - rectDesk.Bottom, 0);

                // Присваиваем значения свойствам формы.

            DesktopBounds = rectNormal;
            WindowState = (FormWindowState) regkey.GetValue(strWinState, 0);
        }
}
```

В предыдущей программе, работавшей с реестром (это DialogsWithRegistry из главы 16), информация загружалась в конструкторе формы и сохранялась в методе *OnClosed*. Однако в NotepadCloneWithRegistry информация загружается в методе *OnLoad*, подключаемом к событию *Load*, Этот метод вызывается после выполнения конструктора, но перед тем как программа станет видимой.

Я выбрал этот вариант, так как хотел, чтобы каждая версия Notepad Clone использовала свой собственный раздел реестра, имя которого формируется на основе имени программы. В конструкторе NotepadCloneWithRegistry выполняется присваивание строки «Notepad Clone with Registry» и полю *strProgName*, и свойству *Text* Следующую версию программы (вы скоро ее увидите) я назвал Notepad-

ClonewithFile. В ней *нолю strProgName* присваивается значение "Notepad Clone with File», а свойству *Text* первоначально присваивается «Notepad Clone with File – Untitled».

На момент вызова метода *OnLoad* класса NotepadCloneWithRegistry все конструкторы по умолчанию уже завершили работу. При выполнении программы NotepadCloneWithRegistry, поле *strProgName* будет равно «Notepad Clone with Registry», а при выполнении NotepadCloneWithFile это поле будет равно «Notepad Clone with File». В методе *OnLoad* поле *strProgName* служит для формирования раздела реестра, который в дальнейшем будет использоваться и в методе *OnClosed*.

Метод *OnLoad* класса NotepadCloneWithRegistry вызывает виртуальный ме год *LoadRegistryInfo*, а метод *OnClosed* — другой виртуальный метод, *SaveRegistryInfo*. Эти методы принимают в качестве параметра *RegistryKey*. Так что любая из пос ледующих версий Notepad Clone может просто переопределить эти методы для загрузки и сохранения информации в реестре. Но переопределенные методы также должны вызывать и базовые методы. Вскоре вы увидите, как все это работ ает.

SaveRegistryInfo сохраняет свойство *WindowState* и четыре компонента поля *rectNormal*. *LoadRegistryInfo* сначала присваивает компонентам *rectNormal* значения, полученные из реестра. Но пользователь мог изменить размер рабочего стола после закрытия предыдущего экземпляра программы. Тогда компоненты *rectNormal* корректируются, так чтобы окно помещалось на рабочем столе. Метод присваивает свойству *DesktopBounds* значение поля *rectNormal* (возможно, скорректированное) и присваивает свойству *WindowState* значение, сохраненное в реестре. Таким образом, при закрытии программы в свернутом или развернутом состоянии при следующем запуске программы устанавливается это состояние. Но тогда восстановления нормального размера окна положение и размер окна становятся такими же, как у предыдущего экземпляра программы перед свертыванием/развертыванием. (Такое поведение немного отличается от поведения Note ad Windows. Notepad всегда при запуске показывается в нормальном состоянии независимо от того, в каком состоянии завершил работу предыдущий экземпляр.)

Клон Notepad с файловым вводом-выводом

Следующий этап разработки Notepad Clone — реализация меню File. Мы уже сталкивались с файловым вводом-выводом. Так как я стараюсь создать программу, пригодную для реальной работы, клон Notepad должен уметь делать то же, что и другие программы, выводящие окно с сообщением «Текст файла был изменен. Сохранить изменения?».

Для реализации этой возможности полезно свойство *TextBoxBase*:

Свойства *TextBoxBase* (выборочно)

Тип	Свойство	Доступ
boo!	*Modified*	Чтение/запись

При создании *TextBox* или *RichTextBox* свойству *Modified* первоначально присваивается значение *false*. В дальнейшем, когда пользователь как-либо изменяет содержимое текстового поля, например вводит символ с клавиатуры, свойству *Modified* поля присваивается *true*. В программах это свойство служит для определения, нужно

ли сохранять содержимое текстового поля в файле. После сохранения содержимого поля в файле программа должна снова присвоить свойству значение *false*. Индикатор изменения текста иногда называют *битом изменения* (dirty bit),

Программа, работающая с документами, должна уметь выводить окно с вопросом, сохранить ли документ. Это происходит, когда пользователь выбирает File | New (что требует прекращения работы с текущим файлом), когда пользователь выбирает File | Open (вместо текущего файла открывается другой) и когда пользователь собирается выйти из программы. В программе NotepadCloneWithFile, которую вы вскоре увидите, метод, проверяющий, можно ли прекратить работу с текущим документом, называется *OkToTrash*.

При выборе пользователем сохранения текущего документа программа обычно делает то же, что и при выборе пользователем File | Save. Однако если у документа еще нет имени, нужно показать диалоговое окно пункта меню Save As. Если пользователь щелкнет в нем кнопку Cancel, файл не сохраняется, но и создание нового файла, открытие файла или выход также не производятся,

Для обработки последней ситуации программа должна переопределять метод *OnClosmg,* выполняемый перед закрытием формы, после этого метода вызывается метод *On-Closed*. Параметром метода *OnClosmg* является объект *CancelEventArgs*, у которого есть единственное свойство *Cancel* типа *bool*. Если в методе *OnClosmg* присвоить этому свойству *true*, программа не закрывается,

Кроме того, для работы NotepadCloneWithFile требуется метод *TextBoxBase,* о котором я еще не говорил:

Методы *TextBoxBase* (выборочно)

```
void ClearUndo()
```

Как я уже упоминал, в классе *TextBoxBase* реализована возможность одноуровневой отмены действий. При внесении изменений в содержимое текстового поля часто сохраняется предыдущая версия. Как мы увидим в следующем разделе, у *TextBoxBase* также имеется свойство *CanUndo* типа *bool*. Если ему задано *true*, значит, имеется предыдущая версия, и к ней можно вернуться, вызвав метод *Undo,*

Однако иногда не нужно, чтобы пользователь мог вернуться к прежнему содержимому текстового поля, например, когда программа загружает содержимое поля из файла. Тогда нет смысла выполнять команду Undo для возврата к прежнему содержимому, которое было до загрузки файла. В таких случаях программа вызывает метод *ClearUndo* для удаления предыдущей версии содержимого поля.

В следующей версии Notepad Clone реализовано меню File. Пока что в пей не реализованы три пункта меню, связанные с печатью. Код для них мы разработаем в главе 21 в программе NotepadCloneWithPrinting.

NotepadCloneWithFile.cs

```
//-------------------------------------------------
// NotepadCloneWithFile.cs © 2001 by Charles Petzold
//-------------------------------------------------
using Microsoft.Win32;              // Для классов работы с реестром
using System;
```

см. след. стр.

```
using System.ComponentModel;        // Для класса CancelEventArgs
using System.Drawing;
using System.IO;
using System.Text;                  // Для класса Encoding
using System.Windows.Forms;

class NotepadCloneWithFile: NotepadCloneWithflegistry
{
                                                                  // Поля
    protected string strFileName;
    const     string strEncoding = "Encoding";      // Для работы с реестром
    const     string strFilter =
                  "Text Documents(*.txt)|*.txt|All Files(*.*)|*.*";
    MenuItem         miEncoding;
    MenuItemEncoding mieChecked;
                                                              // Точка входа
    public new static void Main()
    {
        Application.Run(new NotepadCloneWithFile());
    }
                                                               // Конструктор
    public NotepadCloneWithFile()
    {
        strProgName = "Notepad Clone with File";
        MakeCaption();
        Menu = new MainMenu();

            // Меню File

        MenuItem mi = new MenuItem("&File");
        Menu.MenuItems.Add(mi);
        int index = Menu.MenuItems.Count - 1;

            // File New

        mi = new MenuItem("&New");
        mi.Click += new EventHandler(MenuFileNewOnClick);
        mi.Shortcut = Shortcut.CtrlN;
        Menu.MenuItems[index].MenuItems.Add(mi);

            // File Open

        MenuItem miFileOpen = new MenuItem("&Open...");
        miFileOpen.Click += new EventHandler(MenuFileOpenOnClick);
        miFileOpen.Shortcut = Shortcut.CtrlO;
        Menu.MenuItems[index].MenuItems.Add(miFileOpen);

            // File Save

        MenuItem miFileSave = new MenuItem("&Save");
        miFileSave.Click += new EventHandler(MenuFileSaveOnClick);
```

см. след. стр.

```
miFileSave.Shortcut - Shortcut.CtrlS;
Menu.MenuItems[index].MenuItems.Add(miFileSave);

     // File Save As

mi = new MenuItem("Save &As...");
mi.Click += new EventHandler(MenuFileSaveAsOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

     // File Encoding

ralEncoding = new MenuItem("&Encoding");
Menu.MenuItems[index].MenuItems.Add( miEncoding);
Menu.MenuItems[index].MenuItems.Add("-");

     // Подменю File Encoding

EventHandler eh - new EventHandler(MenuFileEncodingOnClick);

string[] astrEncodings = { "&ASCII", "&Unicode",
                           "&Big-Endian Unicode",
                           "UTF-&7", "&UTF-&8" };

Encoding[] aenc = { Encoding.ASCII, Encoding.Unicode,
                    Encoding.BigEndianUnicode,
                    Encoding.UTF7, Encoding.UTF8 };

for (int i = 0; i < astrEncodings.Length; i++)
{
     MenuItemEncoding mie - new MenuItemEncoding();
     raie.Text = astrEncodings[i];
     mie.Encoding = aenc[i];
     mie.RadioCheck = true;
     mie.Click+=eh;

     miEncoding.MenuItems.AddC mie);
}
mieChecked - (MenuItemEncoding) miEncoding.MenuItems[4];  // UTF-8
mieChecked.Checked = true;

     // File Page Setup

mi = new MenuItem("Page Set&up...");
ffli.Click += new EventHandler(MenuFileSetupOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

     // File Print Preview

mi = new MenuItem("Print Pre&view...");
mi.Click += new EventHandler(MenuFilePreviewOnClick);
```

см. след. стр.

```
        Menu.MenuItems[index].MenuItems.Add(mi);

            // File Print

        mi = new MenuItem("&Print...");
        mi.Click += new EventHandler(MenuFilePrintOnClick);
        mi.Shortcut = Shortcut.CtrlP;
        Menu.MenuItems[index].MenuItems.Add(mi);
        Menu.MenuItems[index].MenuItems.Add("-");

            // File Exit

        mi = new MenuItem("E&xit");
        mi.Click += new EventHandler(MenuFileExitOnClick);
        Menu.MenuItems[index].MenuItems.Add(mi);

            // Установка обработчика системного события.

        SystemEvents.SessionEnding +=
                            new SessionEndingEventHandler(OnSessionEnding);
    }
                                    // Переопределения обработчиков событий
    protected override void OnLoad(EventArgs ea)
    {
        base.OnLoad(ea);

            // Обработка параметров командной строки.

        string[] astrArgs = Environment.GetCommandLineArgs();

        if (astrArgs.Length > 1)      // Первый параметр - имя программы!
        {
            if (File.Exists(astrArgs[1]))
            {
                LoadFile(astrArgs[1]);
            >
            else
            {
                DialogResult dr =
                    MessageBox.Show("Cannot find the " *
                                    Path.GetFileName(astrArgs[1]) *
                                    " file.\r\n\r\n" +
                                    "Do you want to create a new file?",
                                    strProgName,
                                    MessageBoxButtons.YesNoCancel,
                                    MessageBoxIcon.Question);
                switch(dr)
                {
                case DialogResult.Yes:   // Создаем и закрываем файл.
                    File.Create(strFileName - astrArgs[1]).Close();
```

см. след. стр.

```
                    MakeCaption();
                    break;

               case DialogResult.No:
                    break;

               case DialogResult.Cancel:
                    Close();
                    break;
               }
          }
     }
}
protected override void OnClosing(CancelEventArgs cea)
{
     base.OnClosing(cea);

     cea.Cancel = !OkToTrash();
}
                                                  // Обработчики событий
void OnSessionEnding(object obj, SessionEndingEventArgs seea)
{
     seea.Cancel = !OkToTrash();
}
                                                  // Пункты меню
void MenuFileNewOnClick(object obj, EventArgs ea)
{
     if (!OkToTrash())
          return;

     txtbox.Clear();
     txtbox.ClearUndo();
     txtbox.Modified = false;

     strFileName = null;
     MakeCaption();
}
void MenuFileOpenOnClick(object obj, EventArgs ea)
{
     if (!OkToTrash())
          return;

     OpenFileDialog ofd = new OpenFileDialog();
     ofd.Filter = strFilter;
     ofd.FileName = "*.txt";

     if (ofd.ShowDialog() == DialogResult.OK)
          LoadFile(ofd.FileName);
}
void MenuFileEncodingOnClick(object obj, EventArgs ea)
```

См. след.стр.

```
{
    mieChecked.Checked = false;
    mieChecked = (MenuItemEncoding) obj;
    mieChecked.Checked = true;
}
void MenuFileSaveOnClick(object obj, EventArgs ea)
{
    if (strFileName == null || strFileName.Length == 0)
        SaveFileDlg();
    else
        SaveFile();
}
void MenuFileSaveAsOnClick(object obj, EventArgs ea)
{
    SaveFileDlg();
}
protected virtual void MenuFileSetupOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Page Setup not yet implemented!", strProgName);
}
protected virtual void MenuFilePreviewOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Print Preview not yet implemented!", strProgName);
}
protected virtual void MenuFilePrintOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Print not yet implemented!", strProgName);
}
void MenuFileExitOnClick(object obj, EventArgs ea)
{
    if (OkToTrash())
        Application.Exit();
}
                                                    // Переопределения методов
protected override void LoadRegistryInfo(RegistryKey regkey)
{
    base.LoadRegistryInfo(regkey);

        // Присваивание параметров кодировки.

    int Index = (int) regkey.GetValue(strEncoding, 4);

    mieChecked.Checked = false;
    mieChecked = (MenuItemEncoding) miEncoding.MenuItems[index];
    mieChecked.Checked = true;

}
protected override void SaveRegistryInfo(RegistryKey regkey)
{
    base.SaveRegistryInfo(regkey);
```

см. след. а пр.

```
        regkey.SetValue(strEncoding, mieChecked.Index);
    }

    // Вспомогательные методы
    protected void LoadFile(string strFileName)
    {
        StreamReader sr;

        try
        {
            sr = new StreamReader(strFileName);
        y
        catch (Exception exc)
        {
            MessageBox.Show(exc.Message, strProgName,
                            MessageBoxButtons.OK,
                            MessageBoxIcon.Asterisk);
            return;
        1
        txtbox.Text = sr.ReadToEnd();
        sr.Close();

        this.strFileName = strFileName;
        MakeCaption();

        txtbox.SelectionStart - 0;
        txtbox.SelectionLength = 0;
        txtbox.Modified = false;
        txtbox.ClearUndo();
    }

    void SaveFile()
    {
        try
        {
            StreamWriter sw = new StreamWriter(strFileName, false,
                                               mieChecked.Encoding);
            sw.Write(txtbox.Text);
            sw.Close();
        }
        catch (Exception exc)
        {
            MessageBox.Show(exc.Message, StrProgName,
                            MessageBoxButtons.OK,
                            MessageBoxIcon.Asterisk);
            return;
        }
        txtbox.Modified = false;
    }
    bool SaveFileDlg()
    {
```

см. след. стр.

```
        SaveFileDialog sfd = new SaveFileDialog();

        if (strFileName != null && strFileName.Length > 1)
            sfd.FileName = strFileName;
        else
            sfd.FileName = "*.txt";

        sfd.Filter = strFilter;

        if (sfd.ShowDialog() == DialogResult.OK)
        {
            strFileName = sfd.FileName;
            SaveFile();
            MakeCaption();
            return true;
        }
        else
        {
            return false;        // Возвращаемые значения
                                 // используются OkToTrash.
        }
    }
    protected void MakeCaption()
    {
        Text = strProgName + " - " + FileTitle();
    }

    protected string FileTitle()
    {
        return (strFileName != null && strFileName.Length > 1) ?
                    Path.GetFileName(strFileName) : "Untitled";
    }
    protected bool OkToTrash()
    {
        if (!txtbox.Modified)
            return true;

        DialogResult dr =
                MessageBox.Show("The text in the " +' FileTitleO *
                                " file has changed. \r\n\r\n" +
                                "Do you want to save the changes?",
                                strProgName,
                                MessageBoxButtons YesNoCancel,
                                MessageBoxIcon.Exclamation);
        switch (dr)
        {
        case DialogResult.Yes:
            return SaveFileDlg();

        case DialogResult.No:
            return true;
```

см. след. стр.

```
            case DialogResult.Cancel:
                return false;
            }
            return false;
        }
    }
    class MenuItemEncoding: MenuItem
    {
        public Encoding Encoding;
    }
```

Хотя обычно для загрузки файла в Notepad используют пункт меню File | Open. Notepad можно запустить и из командной строки, указав открываемый файл в качестве параметра. Б нашей программе сделано то же самое. NotepadCloneWithFile переопределяет метод *OnLoad* и пытается загрузить *файл*, указанный в качестве параметра командной строки.

В Notepad есть специальное комбинированное окно в диалоговом окне Save As, позволяющее выбрать кодировку файла.



Видите его внизу? Можно выбрать ANSI, Unicode, Unicode Big Endian или UTF-8. (Если вы незнакомы с этими терминами и кодировками текстовых файлов в Windows, прочитайте приложение А.)

К сожалению, программы Windows Forms не столь гибки в отношении диалоговых окон, как программы Windows. Поэтому я поместил пункт Encoding в меню File. Выбор этого пункта выводит подменю с пятью вариантами кодировки: ASCII, Unicode, Big-Endian Unicode, UTF-7 и UTF-8. Небольшой класс *MenuItemEncoding* переопределяет *MenuItem*, так что объект класса *Encoding* можно сохранить вместе с каждым из этих пяти пунктов меню.

NotepadCloneWithFile переопределяет методы *SaveRegistryInfo* и *LoadRegistryInfo* NotepadCloneWithRegistry, позволяя сохранять и затем загружать кодировку символов, выбранную пользователем. Заметьте: эти методы вызывают методы базового класса, поэтому программа по-прежнему сохраняет и загружает положение и размер окна.

Продолжаем разрабатывать клон Notepad

В программе NotepadCloneWithEdit — наследнице NotepadCloneWithFile — я реализовал меню Edit. Сделать это было бы гораздо сложнее и потребовало бы знаний, которые у вас появятся только после прочтения главы 24, если бы не встроенная поддержка операций с буфером обмена классом *TextBoxBase*. Даже в самых первых версиях Notepad Clone можно было вводить Ctrl+Z, Ctrl+C, Ctrl+X и Ctrl+V для выполнения отмены действия, копирования в буфер, копирования в буфер с удалением и вставки из буфера.

Как я уже говорил, отмена действия восстанавливает в поле предыдущую версию текста. Операции копирования и копирования с удалением копируют текст в буфер. Кроме того, при копировании с удалением текст удаляется. Операция вставки копирует текст из буфера в поле. Если в момент выполнения операции вставки выбран какой-то текст, он заменяется текстом, вставленным из буфера,

Хотя *TextBoxBase* поддерживает эти операции, пункты Undo, Cut. Copy и Paste должны присутствовать и в меню Edit. Кроме того, пункт меню Delete (использующий быструю клавишу Delete) должен удалять выделенный текст без копирования в буфер. (В некоторых приложениях для этой цели используется пункт меню Clear.) Довольно интересно, что если просто создать меню Edit с этими пунктами и указать для них соответствующие быстрые клавиши, то текстовое поле просто перестанет реагировать на нажатие этих клавиш. Причина такого поведения в том, что при обработке нажатий на клавиши меню имеет приоритет и если, например, Ctrl+C — быстрая клавиша пункта меню, то нажатие клавиши обрабатывается меню и не доходит до текстового поля,

Заставить эти быстрые клавиши работать с текстовым полем можно, установив обработчики события *Click* для пунктов меню Undo, Cut, Copy, Paste и Delete и вызывая при обработке события следующие методы:

Методы ***TextBoxBase*** (выборочно)

```
void Undo()
void Cut()
void Copy()
void Paste()
void Clear()
```

Кроме того, можно установить обработчик события *Popup* для меню Edit. чтобы активизировать и деактивизировать эти пять пунктов меню в зависимости от ситуации. Пункт Undo можно активизировать в зависимости от значения свойства:

Свойства *TextBoxBase* (выборочно)

Тип	Свойство	Доступ
bool	*CanUndo*	Чтение

Пункты Cut, Copy и Delete (или Clear) активизируются, только если значение свойства *SelectionLength* больше 0. В противном случае нет выделенного текста, который можно удалить или скопировать в буфер.

Осталось свойство Paste, с которым связаны некоторые трудности. Пункт меню Paste активизируется, только если в буфере обмена есть текст. Боюсь, до главы 24

вам придется поверить мне на слово, что пункт Paste должен активизироваться, если следующее выражение возвращает значение *true*:

```
Clipboard.GetDataObject().GetDataPresent(typeof(string))
```

GetDataObject — это статический метод *Clipboard*, возвращающий объект типа *IDataObject*. *GetDataPresent* является методом этого интерфейса, возвращающим *true*, если буфер обмена содержит данные типа, передаваемого в качестве параметра, в данном случае — типа *string*,

Если меню Edit содержало бы только пункты Undo, Cut, Copy, Paste и Delete, все было бы слишком просто. Даже возможность Select All (Выделить все) реализуется вызовом *SelectAll*, а пункт Время и дата (Time/Date) меню Правка (Edit) программы Notepad легко реализовать, используя метод *ToString* класса *DateTime*.

Но меню Edit содержит также пункты Найти (Find), Найти далее (Find Next), Заменить (Replace) и Перейти (Go To). Последний позволяет пользователю перейти на конкретную строку документа. Я решил не делать пункт меню Go To в основном из-за того, что у текстовых полей нет метода, показывающего, на сколько символов отстоит от начала данная строка. (В Win32 API для этой цели служит сообщение *EM_LINEINDEX*.) В качестве компенсации за отсутствие этого пункта меню я реализую то, что не умеет делать Notepad. — выбирать цвет текста и цвет фона.

Б следующем файле реализована работа с диалоговыми окнами Find и Replace. И *FindDialog*, и *ReplaceDialog* наследуют абстрактному классу *FindReplaceDialog*. В этом классе создаются все элементы управления, общие для этих двух классов. Два класса-наследника *FindReplaceDialog* деактивизируют некоторые из этих элементов управления; кроме того, в одном из классов кнопка перемещается в более подходящее место,

FindReplaceDialog.cs

```
//-----------------------------------------
// FindReplaceDialog.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FindDialog: FindReplaceDialog
{
     public FindDialog()
     {
          Text = "Find";

          labelReplace.Visible = false;
          txtboxReplace.Visible = false;
          btnReplace.Visible = false;
          btnReplaceAll.Visible = false;
          btnCancel.Location = btnReplace.Location;
     }
```

см. след. стр.

```
}
class fleplaceDialog : FindReplaceDialog
{
    public ReplaceDialog()
    {
        Text = "Replace";

        grpboxDirection.Visible = false;
    }
}
abstract class FindReplaceDialog: Form
{
                                                    // Поля
    protected Label       labelFind, labelReplace;
    protected TextBox     txtboxFind, txtboxReplace;
    protected CheckBox    chkboxMatchCase;
    protected GroupBox    grpboxDirection;
    protected RadioButton radiobtnUp, radiobtnDown;
    protected Button      btnFindNext, btnReplace, btnReplaceAll, btnCancel;

                                                    // Открытые события
    public event EventHandler FindNext;
    public event EventHandler Replace;
    public event EventHandler ReplaceAll;
    public event EventHandler CloseDlg;
                                                    // Конструктор
    public FindReplaceDialog()
    {
        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox      = false;
        MinimizeBox     = false;
        MaximizeBox     = false;
        ShowInTaskbar   = false;
        StartPosition   = FormStartPosition.Manual;
        Location        = ActiveForm.Location +
                          SystemInformation.CaptionButtonSize +
                          SystemInformation.FrameBorderSize;

        labelFind = new Label();
        labelFind.Parent = this;
        labelFind.Text = "Fund what:";
        labelFind.Location = new Point(8, 8);
        labelFind.Size = new Size(64, 8);

        txtboxFind = new TextBox();
        txtboxFind.Parent = this;
        txtboxFind.Location = new Point(72, 8);
        txtboxFind.Size = new Size(136, 8);
        txtboxFind.TextChanged +=
                    new EventHandler(TextBoxFindOnTextChanged);
```

см. след. < тр.

```
labelReplace = new Label();
labelReplace.Parent = this;
labelReplace.Text = "Re&place with:";
labelReplace.Location = new Point(8, 24);
labelReplace.Size - new Size(64, 8);

txtboxReplace = new TextBox();
txtboxReplace.Parent = this;
txtboxReplace.Location = new Point(72, 24);
txtboxReplace.Size = new Size(136, 8);

chkboxMatchCase = new CheckBox();
chkboxMatchCase.Parent = this;
chkboxMatchCase.Text = "Hatch &case";
chkboxMatchCase.Location = new Point(8, 50); // 48);
chkboxMatchCase.Size = new Size(64, 8);

grpboxDirection = new GroupBox();
grpboxDirection.Parent = this;
grpboxDirection.Text = "Direction";
grpboxDirection.Location = new Point(100, 40);
grpboxDirection.Size = new Size(96, 24);

radiobtnUp = new RadioButton();
radiobtnUp.Parent - grpboxDirection;
radiobtnUp.Text = "&Up";
radiobtnUp.Location = new Point<8, 8);
radiobtnUp.Size = new Size(32, 12);

radiobtnDown = new RadioButton();
radiobtnDown.Parent = grpboxDirection;
radiobtnDown.Text - "&Down";
radiobtnDown.Location = new Point(40, 8);
radiobtnDown.Size = new Size(40, 12);

btnFindNext = new Button();
btnFindNext.Parent   = this;
fatnFindNext.Text =  "&Find  Next";
btnFindNext.Enabled  = false;
btnFindNext.Location = new Point(216, 4);
btnFindNext.Size     = new Size(64, 16);
btnFindNext.Click  += new EventHandler(ButtonFindNextOnClick);

btnReplace = new Button();
btnReplace.Parent   = this;
btnReplace.Text     * "&Replace";
btnReplace.Enabled  = false;
btnReplace.Location - new Point(216, 24);
btnReplace.Size     = new Size(64, 16);
```

см. след. стр.

```
        btnReplace.Click   += new EventHandler(ButtonReplaceOnClick);

        btnReplaceAll = new Button();
        btnReplaceAll.Parent   = this;
        btnReplaceAll.Text     = "Replace &All";
        btnReplaceAll.Enabled  = false;
        btnReplaceAll.Location = new Point(216, 44);
        btnReplaceAll.Size     = new Size(64, 16);
        btnReplaceAll.Click   += new EventHandler(ButtonReplaceAllOnClick);

        btnCancel = new Button();
        btnCancel.Parent   = this;
        btnCancel.Text     = "Cancel";
        btnCancel.Location = new Point(216, 64);
        btnCancel.Size     = new Size(64, 16);
        btnCancel.Click   += new EventHandler(ButtonCancelOnClick);
        CancelButton = btnCancel;

        ClientSize = new Size(288, 84);
        AutoScaleBaseSize = new Size(4, 8);
    }
                                                    // Свойства
    public string FindText
    {
        set { txtboxFind.Text = value; }
        get { return txtboxFind.Text; }
    }
    public string FindReplaceText
    {
        set { txtboxReplace.Text = value; }
        get { return txtboxReplace.Text; }
    }
    public bool MatchCase
    {
        set { chkboxMatchCase.Checked = value; }
        get { return chkboxMatchCase.Checked; }
    }
    public bool FindDown
    {
        set
        {
            if (value)
                radiobtnUp.Checked = true;
            else
                radiobtnDown.Checked = true;
        }
        get { return radiobtnDown.Checked; }
    }
                                                    // Обработчики событий
    void TextBoxFindOnTextChanged(object obj, EventArgs ea)
```

см. след. стр.

```
    {
        btnFindNext.Enabled =
        btnReplace.Enabled =
        btnReplaceAll.Enabled = txtboxFind.Text.Length > 0;
    }
    void fluttonFiftdNextOnCHekC object obj, EventArgs ea)
    {
        if (FindNext I» null)
            FindNext(this, EventArgs.Empty);
    }
    void ButtonReplaceOnClick(object obj, EventArgs «a)
    {
        if (Replace 1* null)
            Replace(this, EventArgs.Empty);
    }
    void ButtonReplaceAllOnClick(object obj, EventArgs ea)
    {
        if (ReplaceAll != null)
            ReplaceAll(this, EventArgs.Empty);
    }
    void tettonCarjoelOrrf31ick(object obj, EventArgs ea)
    {
        if (CloseDlg != null)
            CloseDlg(this, EventArgs.Empty);

        Close();
    }
}
```

Четыре свойства класса *FindReplaceDialog* предоставляют главной программе доступ к двум текстовым полям (тексту, который нужно найти, и тексту, на который нужно заменить), флажку Match Case (чувствительность к регистру) и переключателям, указывающим направление поиска (вверх или вниз по документу).

Как *FindDialog*, так и *ReplaceDialog* — немодальные диалоговые окна. Это классические примеры диалоговых окон, лучше всего работающих в немодальном режиме. При многократном поиске и замене желательно видеть диалоговое окно поверх документа. Как вы видели в главе 16, немодальные диалоговые окна обычно взаимодействуют с основной программой посредством событий. В FindReplaceDialog определены четыре открытых события:

```
public event EventHandler FindNext;
public event EventHandler Replace;
public event EventHandler ReplaceAll;
public event EventHandler CloseDlg;
```

Они происходят, когда пользователь нажимает кнопки диалогового окна Find Next, Replace, Replace All и Cancel.

В программе NotepadCloneWithEdit добавлено меню Edit и реализованы все пункты этого меню.

NotepadCloneWithEdit.cs

```
//-------------------------------------------------
// NotepadCloneWithEdit.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class NotepadCloneWithEdit: NotepadCloneWithFile
{
     MenuItem rolEditUnde. miEditCut, miEditCopy, miEditPaste, miEditDelete;
     string   strFind = "", strReplace - "";
     bool     bMatchCase = false, bFindDown = true;

     public new static void Main()
     {
          Application.Run(new NotepadCloneWithEdit());
     }
     public NotepadCloneWithEdit()
     {
          strProgName = "Notepad Clone with Edit";
          MakeCaption();

               // меню Edit

          MenuItem mi = new MenuItem("&Edit");
          mi.Popup += new EventHandler(MenuEditOnPopup);
          Menu.MenuItems.Add(mi);
          int index = Menu.MenuItems.Count - 1;

               // пункт меню Edit Undo

          miEditUndo = new MenuItem("&Undo");
          miEditUndo.Click += new EventHandler(MenuEditUndoOnClick);
          miEditUndo.Shortcut = Shortcut.CtrlZ;
          Menu.MenuItems[index].MenuItems.Add(miEditUndo);
          Menu.MenuItems[index].MenuItems.Add("-");

               // пункт меню Edit Cut

          miEditCut = new MenuItem("Cu&t");
          fflliEditCut.Click<  +=   new   EventHandler(MenuEditCutOnClick);
          miEditCut.Shortcut = Shortcut.CtrlX;
          Menu.MenuItems[index].MenuItems.Add(miEditCut);

               // пункт меню Edit Copy

          miEditCopy * new MenuItem("&Copy");
          miEditCopy.Click += new EventHandler(MenuEditCopyOnClick);
```

см. след. стр.

```
raiEditCopy. Shortcut = Shortcut.CtrlC;
Menu.MenuItems[index].MenuItems.Add(miEditCopy);

    // пункт меню Edit Paste

miEditPaste = new MenuItem("&Paste");
miEditPaste.Click += new EventHandler(MenuEditPasteOnClick);
miEditPaste.Shortcut * Shortcut.CtrlV;
Menu.MenuItems[index].MenuItems.Add(miEdit Paste);

    // пункт меню Edit Delete

miEditDelete = new MenuItem("De&lete");
mi Edit Delete, Click += new EventHandler(MenuEditDeleteOnClick);
miEditDelete.Shortcut = Shortcut.Del;
Menu.MenuItems[index].MenuItems.Add(miEditDelete);
Menu.MenuItems[index].MenuItems.Add("-");

    // пункт меню Edtt Find

mi = new MenuItem("&Find...");
mi.Click += new EventHandler(MenuEditFindOnClick);
 mi.Shortcut = Shortcut.CtrlF;
Menu.MenuItems[index].MenuItems.Add(mi);

    // пункт меню Edit Find Next

mi = new MenuItem("Find &Next");
mi.Click += new EventHandler(MenuEditFindNextOnClick);
mi.Shortcut = Shortcut.F3;
Menu.MenuItems[index].MenuItems.Add(mi);

    // пункт меню Edit Replace

mi = new MenuItem("&Replace...");
mi.Click += new EventHandler(MenuEditReplaceOnClick);
mi.Shortcut = Shortcut.CtrlH;
Menu.MenuItems[index].MenuItems.Add(mi);
Menu.MenuItems[index].MenuItems.Add("-");

    // пункт меню Edit Select All

mi = new MenuItem("Select &All");
mi.Click += new EventHandler(MenuEditSelectAllOnClick);
mi.Shortcut = Shortcut.CtrlA;
Menu.MenuItems[index].MenuItems.Add(mi);

    // пункт меню Edit Time/Date

mi = new MenuItem("Time/&Date");
```

См. след. стр.

```
        mi.Click += new EventHandler(MenuEditTimeDateOnClick);
        mi.Shortcut = Shortcut.F5;
        Menu.MenuItems[index].MenuItems.Add(mi);
}

void MenuEditOnPopup(object obj, EventArgs ea)
{
        miEditUndo.Enabled - txtbox.CanUndo;

        roiEditCut,Enabled    =
        miEditCopy.Enabled =
        raiEditDelete,Enabled = (txtbox.SelectionLength > 0);

        miEditPaste.Enabled =
             Clipboard.GetDataObject().GetDataPresent(typeof(string));
}
void MenuEditUndoOnClick(object obj, EventArgs ea)
{
        txtbox.Undo();
        txtbox.ClearUndo();
}
void MenuEditCutOnClick(object obj, EventArgs ea)
i
        txtbox.Cut();
}
void MenuEditCopyOnClick(object obj, EventArgs ea)
{
        txtbox.Copy();
}
void MenuEditPasteOnClick(object obj, EventArgs ea)
{
        txtbox.Paste();
}
void MenuEditDeleteOnClick(object obj, EventArgs ea)
{
        txtbox.Clear();
}
void MenuEditFindOnClick(object obj, EventArgs ea)
{
        if (OwnedForms.Length > 0)
             return;

        txtbox.HideSelection = false;

    , FindDialog dlg = new FindDialog();

        dlg.Owner     = this;
        dlg.FindText  = strFind;
        dlg.MatchCase = bMatchCase;
        dlg.FindDown  = bFindDown;
```

см. след. стр.

```
        dlg.FindNext += new EventHandler(FindDialogOnFindNext);
        dlg.CloseDlg += new EventHandler(FindReplaceDialogOnCloseDlg);
        dlg.Show();
    }

    void MenuEditFindNextOnClick(object obj, EventArgs ea)
    {
        if (strFind.Length == 0)
        {
            if (OwnedForms.Length > 0)
                return;

            MenuEditFindOnClick(obj, ea);
        }
        else
            FindNext();
    }
    void MenuEditReplaceOnClick(object obj, EventArgs ea)
    {
        if (OwnedForms.Length > 9)
            return;

        txtbox.HideSelection = false;

        ReplaceDialog dlg = new ReplaceDialog();

        dlg.Owner       * this;
        dlg.FindText    = strFind;
        dlg.ReplaceText = strReplace;
        dlg.MatchCase   * bMatchCase;
        dlg.FindDown    = bFindDown;
        dlg.FindNext   += new EventHandler(FindDialogOnFindNext);
        dlg.Replace    += new EventHandler(ReplaceDialogOnReplace);
        dlg.ReplaceAll += new EventHandler(ReplaceDialogOnReplaceAll);
        dlg.CloseDlg   += new EventHandler(FindReplaceDialogOnCloseDlg);
        dlg.Show();
    }
    void MenuEditSelectAllOnClick(object obj, EventArgs ea)
    {
        txtbox.SelectAll();
    }
    void MenuEditTimeDateOnClick(object obj, EventArgs ea)
    {
        DateTime dt = DateTime.Now;
        txtbox.SelectedText * dt.ToString("t") * " " + dt.ToString("d");
    }
    void FindDialogOnFindNext(object obj, EventArgs ea)
    {
        FindReplaceDialog dlg = (FindReplaceDialog) obj;
```

см. след, стр.

```
        strFind     =tflg.FindText;
        bMatchCase = dlg.MatchCase;
        bFindDown  =  tflg.FindDown;

        FindNext();
}

bool FindNext()
{
        if (bFindDown)
        i
                int iStart - txtbox.SelectionStart + txtbox.SelectionLength;

                while (iStart + strFind.Length <= txtbox.TextLength)
                {
                        if (string.Compare(strFind, 0, txtbox.Text, iStart,
                                        strFind.Length, !bMatchCase) == 0)
                        {
                                txtbox.SelectionStart - iStart;
                                txtbox.SelectionLength - strFind.Length;
                                return true;
                        1
                        iStart++;
                }
        }
        else
        i
                int iStart = txtbox.SelectionStart - strFind.Length;

                while (iStart >= 0)
                i
                        if (string.Compare(strFind, 0, txtbox.Text, iStart,
                                        strFind.Length, ! bMatchCase) == 0)
                        {
                                txtbox.SelectionStart * iStart;
                                txtbox.SelectionLength = strFind.Length;
                                return true;
                        }
                        iStart-;
                }
        }
        MessageBox.Show("Cannot find \"" + strFind + "\"", strProgName,
                        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        return false;
}
void ReplaceDialogOnReplace(object obj, EventArgs ea)
{
        FindReplaceDialog dlg * (FindReplaceDialog) obj;

        strFind     = dlg.FindText;
```

см. след. стр.

```
            strReplace = dlg.ReplaceText;
            bHatchCase = dlg.MatchCase;

            if (string.Compare(strFind, txtbox.SelectedText, !bMatchCase) == 8)
            {
                txtbox.SelectedText = strReplace;
            }
            FindNext();
        }
        void ReplaceDialogOnReplaceAll(object obj, EventArgs ea)
        {
            FindReplaceDialog dlg = (FindReplaceDialog) obj;
            string str = txtbox.Text;
            strFind    = dlg.FindText;
            strReplace = dlg.ReplaceText;
            bMatchCase = dlg.MatchCase;

            if (bHatchCase)
            {
                str = str.Replace(strFind, strReplace);
            }
            else
            {
                for (int i = 0; i < str.Length - strFind.Length; )
                <
                    if (String.Compare(str, i, strFind, 0,
                                       strFind.Length, true) == 0)
                    {
                        str = str.Remove(i, strFind.Length);
                        str = str.Insert(i, strReplace);
                        i += strReplace.Length;
                    }
                    else
                    {
                        i += 1;
                    }
                }
            }
            if (str != txtbox.Text)
            {
                txtbox.Text = str;
                txtbox.SelectionStart = 0;
                txtbox.SelectionLength = 0;
                txtbox.Modified = true;
            }
        }
        void FindReplaceDialogOnCloseDlg(object obj, EventArgs ea)
        {
            txtbox.HideSelection = true;
        }
    }
```

Когда пункт меню служит для вызова немодального диалогового окна, несколько копий диалогового окна обычно не показывают. Поэтому пункты Find и Replace проверяет свойство *OwnedForms*, возвращающее массив объектов *Form*. Если свойство *Length* этого свойства больше 0, значит, немодальное диалоговое окно *уже* выведено, и еще одно показывать не нужно. Как обычно, свойству *Owner* немодальных диалоговых окон присваивается значение *this*, ссылающееся на саму программу.

Когда программа создает одно из двух немодальных диалоговых окон, она, кроме того, устанавливает обработчики событий, определенных в диалоговых окнах. Большая часть выполняемой программой работы заключается в реакции на эти события. Для реализации логики поиска в программе служит метод *Compare* класса *String*, а для замены текста — *Remove* и *Insert* (эти методы описаны в приложении B).

Следующая версия Notepad Clone относительно проста. В ней реализовано меню Format, расширяющее возможности Notepad. Наряду со стандартными пунктами Word Wrap и Font в нем имеется Background Color. Word Wrap — это обычный пункт меню с галочкой, тогда как Font и Background Color показывают диалоговые окна *FontDialog* и *ColorDialog*.

NotepadCloneWithFormat.cs

```
//-----------------------------------------------------
// NotepadCloneWithFormat.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Windows.Forms;

class NotepadCloneWithFormat : NotepadCloneWithEdit
{
          // Строки для работы с реестром

     const string strWordWrap  = "Wordwrap";
     const string strFontFace  = "FontFace";
     const string strFontSize  = "FontSize";
     const string strFontStyle = "FontStyle";
     const string strForeColor = "ForeColor";
     const string strBackColor = "BackColor";
     const string strCustomClr = "CustomColor";

     ColorDialog clrdlg = new ColorDialog();
     MenuItem    miFormatWrap;

     public new static void Main()
     {
          Application.Run(new NotepadCloneWithFormat());
     }
     public NotepadCloneWithFormat()
     {
```

см. след. стр.

```
        strProgName = "Notepad Clone with Format";
        MakeCaption();

            // Format

        MenuItem ml = new MenuItem("F&ormat");
        mi. Popup += new EventHandler(MenuFormatOnPopup);
        Menu.MenuItems.Add(mi);
        int index - Menu.MenuItems.Count - 1;

            // Format Word Wrap

        rat FormatWrap = new MenuItem("&Word Wrap");
        miFormatWrap.Click += new EventHandler(MenuFormatWrapOnClick);
        Menu.MenuItems[index].MenuItems.Add(miFormatWrap);

            // Format Font

        mi = new MenuItem("&Font...");
        mi.Click += new EventHandler(MenuFormatFontOnClick);
        Menu.MenuItems[index].MenuItems.Add(mi);

            // Format Background Color

        mi = new MenuItem("Background &Color...");
        mi.Click += new EventHandler(MenuFormatColorOnClick);
        Menu.MenuItems[index].MenuItems.Add(mi);
    }
    protected override void OnLoad(EventArgs ea)
    {
        base.OnLoad(ea);

            // Help

        MenuItem ffli = new MenuItem("&Help");
        Menu.MenuItems.Add(mi);
        int index = Menu.MenuItems.Count - 1;

            // Help About

        mi = new MenuItem("&About " + strProgName + "...");
        mi.Click += new EventHandler(MenuHelpAboutOnClick);
        Menu.MenuItems[index].MenuItems.Add(mi);
    }
    void MenuFormatOnPopup(object ob], EventArgs ea)
    {
        miFormatWrap.Checked = txtbox.WordWrap;
    }
    void MenuFormatWrapOnClick(object obj, EventArgs ea)
    {
```

см. след. стр.

```
        MenuItem mi = (MenuItem) obj;
        ffli.Checked^=true;
        txtbox.WordWrap * mi.Checked;
}
void MenuFormatFontOnClick(object obj, EventArgs ea)
{
        FoRtDialog fontdlg * new FontDialog();

        fontdlg.ShowColor = true;
        fontdlg.Font = txtbox.Font;
        fontdlg.Color = txtbox.ForeColor;

        if (fontdlg.ShowDialog() == DialogResult.OK)
        {
            txtbox.Font = fontdlg.Font;
            txtbox.ForeColor = fontdlg.Color;
        }
}
void MenuFormatColorOnClick(object obj, EventArgs ea)
<
        clrdlg.Color = txtbox.BackColor;

        if (clrdlg.ShowDialog() == DialogResult.OK)
            txtbox.BackColor = clrdlg.Color;
}
void MenuHelpAboutOnClick(object obj, EventArgs ea)
{
        MessageBox.Show(strProgName + " $ 2001 by Charles Petzold",
                        strProgName);
}
protected override void LoadRegistryInfo(RegistryKey regkey)
{
        base.LoadRegistryInfo(regkey);

        txtbox.WordWrap = Convert.ToBoolean(
                                (int) regkey.GetValue(strWordWrap));
        txtbox.Font = new Font((string) regkey.GetValue(strFontFace),
                               float.Parse(
                                 (string) regkey.GetValue(strFontSize)),
                               (FontStyle) regkey.GetValue(strFontStyle));
        txtbox.ForeColor = Color.FromArgb(
                                 (int) regkey.GetValue(strForeColor));
        txtbox.BackColor = Color.FromArgb(
                                 (int) regkey.GetValue(strBackColor));

        int[] aiColors = new int[16];

        for (int i = 0; i < 16; i++)
            aiColors[i] = (iRt) regkey.GetValue(strCustomClr + i);
```

см. след, стр.

```
            clrdlg.CustomColors = aiColors;
        }
        protected override void SaveRegistryInfo(RegistryKey regkey)
        {
            tease.$ave8egistrylRfo(regkey);

            regkey.SetValue(strWordWrap,  Convert.ToInt32(txtbox.WordWrap));
            regkey.SetValue(strFontFace,  txtbox.Font.Name);
            regkey.SetValue(strFontSize,  txtbox.Font.SizeInPoints.ToString());
            regkey.SetValue(strFontStyle, (int) txtbox.Font.Style);
            regkey.SetValue(strForeColor, txtbox.ForeColor.ToArgb());
            regkey.SetValue(strBackColor, txtbox, BackColor.ToArgb());

            for (int i = 0; i < 16; i++)
                regkey.SetValue(strCustomClr + i, clrdlg.CustomColors[i]);
        }
    }
```

Кроме того, в этой версии программы реализовано меню Help, содержащее пункт About. Я поместил создание этого пункта в очередное переопределение метода *OnLoad*. Это позволяет поместить в пункт About значение поля *strProgName*, показывающее имя программы. В случае создания пункта About в конструкторе у любой программы, которая является ее потомком, имя программы в пункте меню было бы неправильным,

Все параметры, указываемые в меню Format, сохраняются в реестре. Для этого программа переопределяет методы *LoadRegistryInfo* и *SaveRegistryInfo*.

В главе 21 я еще более усовершенствую Notepad Clone, реализовав три пункта меню, отвечающие за печать (программа будет называться NotepadCloneWithPrinting). В главе 24 я завершу работу над программой, добавив возможность буксировки. В последней версии — NotepadClone — можно будет буксировать в программу файлы или текст из других приложений.

Использование текстовых полей в специальных целях

Чаще всего текстовые поля служат для ввода и редактирования текста пользователем. Но есть и еще несколько специальных применений текстовых полей, определяемых следующими свойствами:

Свойства *TextBox* (выборочно)

Тип	Свойство	Доступ
char	*PasswordChar*	Чтение/запись
CharacterCasing	*CharacterCasing*	Чтение/запись
bool	*Readonly*	Чтение/запись

Текстовое поле можно задействовать, когда пользователь должен ввести пароль или другие данные, которые нужно скрыть от остальных. Обычно поле *PasswordChar* равно 0, но если присвоить ему другое значение, то любой символ, вводи-

мый пользователем, будет показываться этим символом. Обычно в окнах ввода паролей в качестве такого символа используется «звездочка». Чтобы свойство *PasswordChar* работало, свойство *Multiline* должно иметь значение *false,*

Иногда (например, тоже при вводе пароля), нужно преобразовать текст в верхний или нижний регистр. Для этого используется свойство *CharacterCasing.* Ему можно присвоить одно из следующих значений;

Перечисление *CharacterCasing*

Член	Значение
Normal	О
Upper	\
Lower	*1*

Свойство *Readonly* определено в *TextBoxBase* и используется и в элементах управления *RichTextBox. Если* ему задано *true,* пользователь не может ничего ввести в текстовое поле. Но при этом в текстовом поле по-прежнему присутствует каретка, текст можно выделить и скопировать в буфер обмена. Неизменяемое текстовое поле — отличный выбор для программ, которые должны показывать пользователю текстовую информацию, особенно когда количество информации заранее неизвестно. Элемент управления *Label* отлично подходит для работы с короткими строками, тогда как *TextBox* позволяет пользователю прокручивать большие блоки текста. Дополнительным преимущество — возможность копирования текста из поля в буфер обмена. В следующей программе этой главы (EnumerateEnumeration) я использую неизменяемое текстовое поле.

Поле ввода с форматированием

Боюсь, я не смогу подробно рассказать о многих возможностях *RichTextBox,* так как тогда этот раздел слишком разрастется. Название свое этот класс получил по двум причинам. Во-первых, он поддерживает *форматированный текст* (rich text), в котором могут применяться разные шрифты, отступы абзацев и табуляция. Во-вторых, *RichTextBox* может импортировать/экспортировать текст в соответствии со спецификацией Rich Text Format (RTF). RTF разработан Microsoft примерно в 1986 г. для обмена форматированным текстом между Windows-приложениями, Документацию по RTF 1.6 см. по адресу *http://msdn.microsoft.com/library/specs/rtfspec.htm.*

Документ формата RTF — этот текстовый файл, в котором обратной косой чертой (\) обозначаются тэги форматирования (например, \i означает начало блока текста, помеченного курсивом), а фигурные скобки { и } служат для группировки тэгов. Формат RTF существует уже давно и поддерживается многими текстовыми процессорами. Однако на данном этапе развития форматирования текста самая большая проблема RTF в том. что это не HTML.

Аналогично тому, как Notepad построен на основе используемого в Win32 эквивалента элемента управления *TextBox,* WordPad построен на основе используемого в Win32 эквивалента *RichTextBox.* В элементе управления *TextBox* можно указать цвет или шрифт только для всего содержимого поля, а в *RichTextBox* можно определить шрифты, цвета, способы выравнивания абзацев, отступы и т. д. Все это

форматирование указывается для текущего выделения с помощью свойств *SelectionFont, SelectionColor, SelectionAlignment, SelectionIndent* и т. д.

Для доступа к данным RTF используются свойства:

Свойства *RichTextBox* (выборочно)

Тип	Свойство	Доступ
string	*Rtf*	Чтение/запись
string	*SelectedRtf*	Чтение/запись

Кроме того, *RichTextBox* содержит два метода (каждый имеет три перегруженных версии), позволяющих загружать файл в элемент управления (*LoadFile*) и сохранять документ в файле (*SaveFile*).

Подсказки

Подсказка (ToolTip) — небольшое прямоугольное окно с пояснительным текстом. Этот текст обычно показывается, когда курсор мыши помещается поверх элемента управления. Подсказки реализованы в классе *ToolTip* библиотеки Windows Forms. Хотя подсказки Windows Forms обладают меньшими возможностями в сравнении с подсказками Win32 API, они гораздо проще в использовании.

Для создания подсказок к набору элементов управления достаточно определить один объект *ToolTip*:

```
ToolTip tooltip = new ToolTip();
```

Обычно *ToolTip* определяется для формы (если форма содержит элементы управления) и в конструкторе каждого диалогового окна, создаваемого программой. Если на форме, кроме панели инструментов и строки состояния, нет элементов управления, объект *ToolTip* не нужен, так как у панели инструментов и строки состояния есть собственные средства работы с подсказками. (О панелях инструментов и строках состояния см. главу 20.)

ToolTip — один из немногих классов, в которых несколько методов значительно важнее свойств и событий:

Методы *ToolTip* (выборочно)

```
void SetToolTip(Control cntl, string strTip)
string GetToolTip(Control cntl)
void RemoveAll()
```

Для объекта *ToolTip,* создаваемого для определенного диалогового окна, нужно вызвать метод *SetToolTip* по разу для каждого элемента управления, для которого указывается текст подсказки, например:

```
ToolTip tooltip = new ToolTipO;
tooltip.SetToolTip(btnBigger, "This button increases the font size");
tooltip.SetToolTip(btnSmaller, "This button decreases the font size");
```

Чтобы текст подсказки появлялся на нескольких строках, используйте символы перевода строки (‘\n’). Для длинных подсказок я обычно вставляю символы перевода строки примерно через каждые 32 символа.

Чтобы удалить подсказку для элемента, присвойте тексту значение *null:*

```
tooltip.SetToolTip(btn, null)
```

Все подсказки удаляются так:

```
tooltip.RemoveAll()
```

Получить список всех элементов, для которых определены подсказки, нельзя.

Следующие свойства *ToolTip* определяют видимость текста подсказки:

Свойства *ToolTip* (выборочно)

Тип	Свойство	Доступ	Значение по умолчанию
bool	*Active*	Чтение/запись	*true*
boot	*ShowAlways*	Чтение/запись	*false*

Значение *false* свойства *Active* отключает показ текста подсказки для всех элементов управления, связанных с объектом *ToolTip.* Значение *true* свойства *ShowAlways* приводит к тому, что текст подсказки показывается, даже если родительская форма элемента в данный момент неактивна,

Чтобы подсказки не мельтешили на экране, их обычно не показывают сразу. Подсказка появляется после того, как курсор мыши остановился на элементе (по умолчанию через 0,5 секунды). По истечении другого промежутка времени (по умолчанию 5 секунд) подсказка исчезает с экрана. За это отвечают свойства:

Свойства *ToolTip* (выборочно)

Тип	Свойство	Доступ	Значение по умолчанию
int	*AutomaticDelay*	Чтение/запись	500 (миллисекунд)
int	*InitialDelay*	Чтение/запись	*AutomaticDelay*
int	*ReshowDelay*	Чтение/запись	*AutomaticDelay / 5*
int	*AutoPopDelay*	Чтение/запись	*10* X *AutomaticDelay*

При присвоении значений свойству *AutomaticDelay* остальным трем свойствам автоматически присваиваются значения, указанные в колонке «Значение по умолчанию». Идея в том, что все периоды времени при изменении одного свойства изменяются пропорционально. Но затем можно изменить эти три свойства независимо друг от друга. Остальные три свойства напрямую определяют параметры показа подсказки:

- *InitialDelay* указывает время от момента остановки курсора до момента вывода текста подсказки;
- *ReshowDelay* — период времени перед выводом новой подсказки при перемещении курсора с одного элемента управления на другой;
- *AutoPopDelay* указывает длительность показа подсказки.

Рассмотрим программу, в которой есть три однострочных текстовых поля, многострочное неизменяемое текстовое поле и подсказки, облегчающие жизнь пользователя. Хотя эта программа — один из всего лишь двух примеров работы с подсказками в этой книге, вам, вероятно, придется использовать подсказки везде, где применяются элементы управления.

В таблицах с перечислениями, определенными в библиотеке классов .NET Framework, я обычно указываю числовые значения, связанные с членами перечислений. Возможно, вас заинтересовало, откуда взялись эти таблицы, — ведь числовые значения в документации .NET не приводятся. Получить эти таблицы мне помогла программа, которую я написал на ранней стадии изучения .NET и в результате усовершенствования которой я получил следующий фрагмент кода, названный EnumerateEnumeration.

EnumerateEnumeration.cs

```
//-----------------------------------------------------------
// EnumerateEnumeration.cs © 2001 by Charles Petzold
//-----------------------------------------------------------
using System;
using System.Drawing;
using System.Reflection;                // Для класса Assembly
using System.Text;                      // Для класса StringBuilder
using System.Windows.Forms;

class EnumerateEnumeration: Form
{
     Button   button;
     TextBox  tbLibrary, tbNamespace, tbEnumeration, tbOutput;
     CheckBox cbHex;

     public static void Main()
     {
          Application.Run(new EnumerateEnumeration());
     }

     public EnumerateEnumeration()
     {
          Text = "Enumerate Enumeration";
          ClientSize = new Size(242, 164);

          Label label     = new Label();
          label.Parent    = this;
          label.Text      = "Library:";
          label.Location = new Point(8, 8);
          label.Size      = new Size(56, 8);

          tbLibrary           = new TextBox();
          tbLibrary.Parent    = this;
          tbLibrary.Text      = "system.windows.forms";
          tbLibrary.Location = new Point(64, 8);
          tbLibrary.Size      = new Size(120, 12);
          tbLibrary.Anchor   |= AnchorStyles.Right;

          ToolTip tooltip = new ToolTip();
          tooltip.SetToolTip(tbLibrary,
```

См. след. стр.

```
                        "Enter the name of a .NET dynamic-\n" +
                        "link libary, such as 'mscorlib',\n" *
                        "'system.windows.forms', or\n" +
                        "'system.drawing'.");

label          - new Label();
label.Parent   - this;
label.Text     = "Namespace:";
label.Location * new Point(8, 24);
label.Size     = new Size(58, 8);

tbNamespace       ,, = new TextBox();
tbNamespace.Parent   = this;
tbNamespace.Text     = "System.Windows.Forms";
tbNamespace.Location = new Point(64, 24);
tbNamespace.Size     = new Size(120, 12);
tbNamespace.Anchor   |= AnchorStyles.Right;

tooltip.SetToolTip(tbNamespace,
                        "Enter the name of a namespace\n" +
                        "within the library, such as\n" +
                        "'System', 'System.IO',\n" +
                        "'System.Drawing',\n" +
                        "'System.Drawing.Drawing2D',\n" +
                        "or 'System.Windows.Forms'.");

label          = new Label();
label.Parent   = this;
label.Text     = "Enumeration:";
label.Location = new Point(8, 40);
label.Size     = new Size(56, 8);

tbEnumeration          = new TextBox();
tbEnumeration.Parent   = this;
tbEnumeration.Text     = "ScrollBars";
tbEnumeration.Location = new Point(64, 40);
tbEnumeration.Size     = new Size(120, 12);
tbEnumeration.Anchor   j= AnchorStyles.Right;

tooltip.SetToolTip(tbEnumeration,
                        "Enter the name of an enumeration\n" +
                        "defined in the namespace.");

cbHex                = new CheckBox();
cbHex.Parent         - this;
cbHex.Text           = "Hex";
cbHex.Location       = new Point(192, 16);
cbHex.Size           = new Size(40, 8);
cbHex.Anchor         = AnchorStyles.Top | AnchorStyles.Right;
cbHex.CheckedChanged += new EventHandler(CheckBoxOnCheckedChanged);
```

см. след. стр.

```
        tooltip.SetToolTip(cbHex, "Check this box to display the\n" +
                                  "enumeration values in hexadecimal.");

        button          = new Button();
        button.Parent   = this;
        button.Text     * "OK";
        button.Location = new Point(192, 32);
        button.Size     = new Size(40, 16);
        button.Anchor   = AnchorStyles.Top \ AnchorStyles.Right;
        button.Click   += new EventHandler(ButtonOkOnClick);

        AcceptButton * button;

        tooltip.SetToolTip(button,
                                 "Click this button to display results.");

        tbOutput            - new TextBox();
        tbOutput.Parent     = this;
        tbOutput.ReadOnly   = true;
        tbOutput.Multiline  = true;
        tbOutput.ScrollBars = ScrollBars.Vertical;
        tbOutput.Location   = new Point(8, 56);
        tbOutput.Size       = new Size(226, 100);
        tbOutput.Anchor     = AnchorStyles.Left | AnchorStyles.Top |
                              AnchorStyles.Right | AnchorStyles.Bottom;

        AutoScaleBaseSize = new Size(4, 8); -

            // Инициализация отображения.

        ButtonOkOnClick(button, EventArgs.Empty);
    }
    void CheckBoxOnCheckedChanged(object sender, EventArgs ea)
    {
        button.PerformClick();
    }
    void ButtonOkOnClick(object sender, EventArgs ea)
    {
        FillTextBox(tbOutput, tbLibrary.Text, tbNamespace.Text,
                    tbEnumeration.Text, cbHex.Checked);
    }
    public static bool FillTextBox(TextBox tbOutput, string strLibrary,
                                   string strNamespace,
                                   string strEnumeration, bool bHexadecimal)
    {
        string strEnumText = strNamespace + "." + strEnumeration;
        string strAssembly;

        try
        {
```

см. след. стр.

```
        strAssembly =
            Assembly.LoadWithPartialName(strLibrary).FullName;
    }
    catch
    {
        return false;
    }
    string strFullText = strEnumText + "," + strAssembly;

        // Получаем тип перечисления.

    Type type = Type.GetType(strFullText, false, true);

    if(type == null)
    {
        tbOutput.Text = "V" + strFullText +
                        "\" is not a valid type.";
        return false;
    }
    else if(!type.IsEnum)
    {
        tbOutput.Text = "\"" + strEnumText +
                        "\" is a valid type but not an enum.";
        return false;
    }

        // Получаем все члена этого перечисления.

    string[] astrMembers = Enum.GetNames(type);
    Array    arr         = Enum.GetValues(type);
    object[] aobjMembers = new object[arr.Length];

    arr.CopyTo(aobjMembers, 0);

        // Создаем StringBuilder для текста.

    StringBuilder sb = new StringBuilder();

        // Добавляем имя перечисления и заголовки.

    sb.Append(strEnumeration);
    sb.Append(" Enumeration\r\nMember\tValue\r\n");

        // Добавляем текстовое представление и числовые значения.

    for (int i = 0; i < astrMembers.Length; i++)
    {
        sb.Append(astrMembers[i]);
        sb.Append("\t");
```

см. след. стр.

```
            if (bHexadecimal)
                sb.Append("0x" + Enum.Format(type, aobjMembers[i], "X"));
            else
                sb.Append(Enum.Format(type, aobjMembers[i], "D"));

            sb.Append("\r\n");
        }
            // Добавляем некоторую дополнительную информацию.

        sb.Append("\r\nTotal = " + astrMembers.Length + "\r\n");
        sb.Append("\r\n" * type.AssemblyQualifiedName + "\r\n");

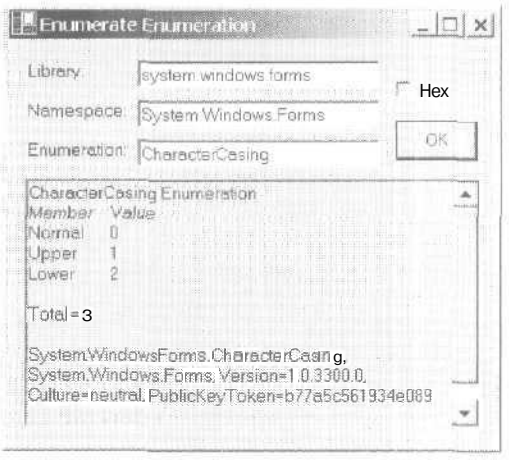
            // Присваиваем текстовому полю свойство Text StringBuilder.

        tbOutput.Text = sb.ToString();
        tbOutput.SelectionLength = 0;
        return true;
    }
}
```

В конструкторе создаются три пары, состоящие из надписи и однострочного текстового поля, предназначенные для ввода пользователем имени DLL, пространства имен из этой DLL и перечисления из этого пространства имен. Флажок позволяет установить показ результатов в шестнадцатеричном представлении. Нажатие кнопки сообщает программе, что ввод исходных данных окончен и нужно показать результаты. Каждому из этих элементов управления соответствует своя подсказка.

Результаты показываются в неизменяемом текстовом поле. Заметьте: для обеспечения гибкости размера всех текстовых полей используется свойство *Anchor*. Если сделать форму шире, все три однострочных текстовых поля станут шире. Если сделать форму выше или ниже, соответствующим образом изменится высота текстового поля только для чтения,

Так выглядела программа при последнем использовании:



Заметьте: у неизменяемого текстового поля не такой цвет фона по умолчанию, как у обычных полей. Так как неизменяемые текстовые поля поддерживают каретку и интерфейс с буфером обмена, я могу выделить текст в поле и, нажав Ctrl+C, скопировать его в буфер обмена. Затем я могу вставить его в документ Microsoft Word и преобразовать в таблицу Word.

Немного поработав с программой, вы почувствуете раздражение от необходимости постоянно набирать имена часто используемых библиотек и пространств имен. Ниже я покажу еще одну версию программы — EnumerateEnumerationCombo, где используются комбинированные окна, работающие с реестром Windows для запоминания всех допустимых комбинаций имен библиотек, пространств имен и перечислений.

Весь код показа информации вынесен в статический метод *FillTextBox* потому, что надо приспособиться к потребностям программы EnumerateEnumerationCombo. Этот метод является статическим, так как EnumerateEnumerationCombo, которая будет его использовать, не будет наследником EnumerateEnumeration. *FillTextBox* возвращает *true,* если комбинация трех текстовых строк допустима, иначе возвращает *false.* EnumerateEnumeration эти сведения не нужны, а EnumcratcEnumeration-Combo — требуются.

FillTextBox использует метод *GetType* класса *Type,* чтобы получить объект *Type* для перечисления. Параметром *GetType* является текстовая строка:

```
namespace.enumeration,library
```

Заметьте: пространство имен и имя перечисления отделены, как обычно, точкой, а перед именем библиотеки стоит запятая. Имя библиотеки должно содержать сведения о версии, поэтому вызывается *Assembly.LoadWithPartialName.* Программа получает имена и значения членов перечисления, вызывая статические методы *GetNames* и *GetValues* класса *Enum.*

Взаимосвязь между именем библиотеки и пространством имен несколько запутана. Так, библиотека system.drawing.dll содержит пространство имен *System.Drawing.* Однако *System.Drawing.Drawing2D* тоже размещается в system.drawing.dll, Многие из основных пространств имен, например *System* и *System.IO,* находятся в mscorlib.dll, что означает «Microsoft Core Library».

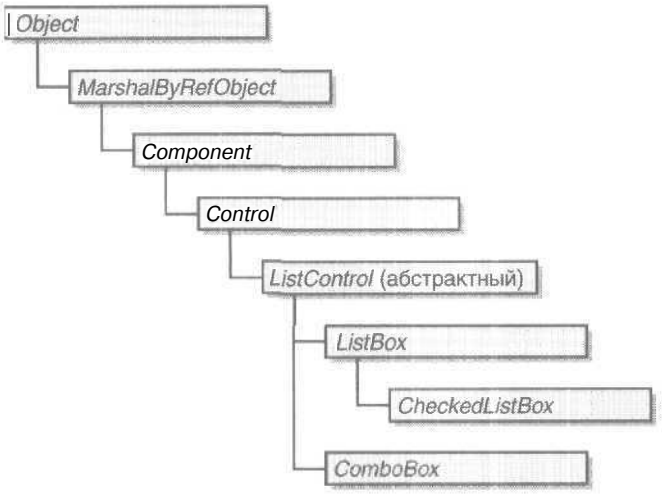
Обратите внимание на использование класса *StringBuilder* для формирования строки, показываемой в текстовом поле только для чтения. (О *StringBuuder* см. приложение В.) В исходной версии текстовое поле заполнялось строка за строкой при помощи оператора +=. Я начал разочаровываться в таком подходе, когда на моем маломощном компьютере при использовании предварительного выпуска Visual Studio .NET показ перечисления *EmfPlusRecordType* занял 30 секунд. Переход на *StringBuuder* сделал изменение поля мгновенным.

Окно списка

Элемент управления *ListBox* часто используют в тех же целях, что и группу переключателей, — чтобы пользователь мог выбрать один элемент из нескольких. Однако списки занимают меньше места на экране и позволяют выбирать несколько элементов. Элемент управления *ComboBox* (о нем я расскажу ниже) обычно еще

компактнее и часто, как и *TextBox*, содержит область, в которую пользователь может вводить информацию.

И *ListBox*, и *ComboBox* являются наследниками *ListControl*:



Обычно после создания списка требуется заполнить его элементами. За это отвечает свойство *Items*:

Свойства *ListBox* (выборочно)

Тип	Свойство	Доступ
ListBox.ObjectCollection	*Hems*	Чтение

Класс *ObjectCollection* — это знакомая реализация интерфейсов *Hist*, *ICollection* и *IEnumerable*. Можно обращаться к элементам *Items* по индексу, как и к элементам массива, использовать *Add* и *AddRange* для добавления элементов в список и искать элементы при помощи методов *Contains* и *IndexOf*.

Элементы *ListBox* имеют тип *Object* и не должны обязательно быть строками. Каждый из объектов, входящих в *ListBox*, отображается при помощи своего метода *ToString*. Конечно, обращаясь к объекту из списка, программа несет ответственность за приведение объекта к соответствующему типу.

Список может содержать несколько одинаковых элементов. Так. если добавить в список элемент, уже имеющийся в наборе, набор будет содержать две копии элемента. Так как это будет смущать пользователя, делать так, пожалуй, не стоит,

При добавлении элемента в список каждому элементу присваивается индекс; индексы нумеруются, начиная с 0. Индекс определяет положение элемента в списке. Обычно индексы возрастают по мере добавления элементов. Но если свойству *Sorted* задать *true*, индексы будут возрастать в соответствии с алфавитным порядком элементов.

Свойства *ListBox* (выборочно)

Тип	Свойство	Доступ
bool	*Sorted*	Чтение/запись
int	*TopIndex*	Чтение/запись

Свойство *TopIndex* содержит индекс элемента, показываемого в начале списка, По умолчаню *TopIndex* равно 0.

Свойство *PreferredHeigh* содержит высоту списка, необходимую для размещениия всех элементов;

Свойства *ListBox* (выборочно)

Тип	Свойство	Доступ
int	*Preferredeight*	Чтение
bool	*IntegralHeight*	Чтение/запись

Обычно высоте списка не задают значение *PreferredHeight.* Свойство *IntegralHeight* по умолчанию имеет значение *true,* что означает установку такой высоты списка, при которой нет элементов, показываемых частично.

Если высоты списка недостаточно, чтобы вместить все элементы, показывается вертикальная полоса прокрутки. Полосу прокрутки можно показывать независимо от количества элементов. Можно показывать и горизонтальную полосу для элементов, ширина которых больше ширины списка:

Свойства *ListBox* (выборочно)

Тип	Свойство	Доступ
bool	*ScrollAlwaysVisible*	Чтение/запись
bool	*HorizontalScrollbar*	Чтение/запись

Следующие свойства *ListBox* отвечают за выделение элементов:

Свойства *ListBox* (выборочно)

Тип	Свойство	Доступ
SelectionMode	*SelectionMode*	Чтение/запись
int	*SelectedIndex*	Чтение/запись
object	*SelectedItem*	Чтение/запись
ListBox.SelectedIndexCollection	*SelectedIndices*	Чтение
ListBox.SelectedObjectCollection	*SelectedItems*	Чтение

Значениями свойства *SelectionMode* являются члены перечисления *SelectionMode:*

Перечисление ***SelectionMode***

Член	Значение
None	0
One	1
MultiSimple	2
MultiExtended	3

Значение по умолчанию — *One.* В любое время выделяется только один из элементов списка, показываемый обратным цветом. Когда список получает фокус ввода, этот же элемент окружается точечной линией — прямоугольником фокуса, аналогичным прямоугольнику фокуса на кнопках. Но увидеть прямоугольник фокуса в данном случае несколько сложнее, так как у него такой же размер, как и у прямоугольника выделения обратным цветом. Можно выбирать элементы мышью,

По умолчанию в только что созданном и заполненном списке никакой элемент не выделен: *SelectedIndex* равно -1, а *SelectedItem* — *null* Используя одно из этих двух свойств, можно после создания списка выбрать определенный элемент,

Свойство *MultiSimple* позволяет обеспечивает возможность выбора нескольких элементов списка. Каждый выделенный элемент показывается обратным цветом. При этом прямоугольник фокуса не зависит от прямоугольника выделения. Можно передвигать прямоугольник фокуса с точечной границей по элементам списка клавишами управления курсором. Нажатие пробела приводит к выделению (или отмене выделения) элемента, отмеченного прямоугольником фокуса. Кроме того, можно выбрать элемент (или отменить выбор элемента) мышью, при этом элемент также получает фокус.

Список, для которого установлен режим выделения *MultiExtended*, на первый взгляд, выглядит как список с выделением одного элемента. Клавиши управления курсором перемешают и прямоугольник выделения, и прямоугольник выбора, Однако можно выделить сразу несколько элементов, если при нажатии клавиш управления курсором удерживать клавишу Shift. Но если ее отпустить и нажать клавишу управления курсором или щелкнуть элемент, это снова приведет к выделению только одного элемента. Кроме того, в списке *MultiExtended*можно выбирать отдельные элементы (или отменять выбор элементов) щелчком при нажатой клавише Ctrl.

Пожалуй, применять список *MultiExtended*имеет смысл, когда предполагается, что пользователь будет выбирать *диапазон* элементов, а *MultiSimple* — когда предполагается, что пользователь будет выбирать непоследовательные элементы

В списке с выделением одного элемента для получения индекса выделенного элемента или самого выделенного элемента служат свойства *SelectedIndex* или *SelectedIlem*. Если *Istbox* — объект типа *ListBox*, то:

```
lstbox.SelectedItem
```

равносильно:

```
lstbox.Items[lstbox.SelectedIndex]
```

Кроме того, можно задействовать свойство *Text* для получения текстового представления выделенного элемента. Выражение:

```
lstbox.Text
```

равносильно:

```
lstbox.SelectedItem.ToString()
```

В списке с выделением нескольких элементов свойства *SelectedIndices* и *SelectedItems* предоставляют доступ к выделенным элементам. И у класса *SelectedIndexCollection*, и у класса *SelectedObjectCollection*имеются свойство *Count* и индексатор, *Count* показывает количество выделенных элементов и, как обычно, индексатор находится в диапазоне от 0 до (*Count*–1). Индексатор объекта *SelectedIndices* возвращает индекс выделенного элемента списка. Так, если *index* находится в диапазоне от 0 до (*Count*–1),то:

```
lstbox.SelectedItems[index]
```

равносильно:

```
lstbox.Items[lstbox.SelectedIndices[index]]
```

Чтобы установить первоначальное выделение в списке с выделением нескольких элементов, вызывается метод *SetSelected* для каждого выделяемого элемента:

Методы *ListBox* (выборочно)

```
void SetSelected(int index, bool bSelect)
bool GetSelected(int index)
void ClearSelected()
```

Кроме того, метод *GetSelected* возвращает *true* для каждого индекса, соответствующего индексу выделенного элемента. *ClearSelected* отменяет выделение всех элементов списка.

Часто при использовании списков в диалоговых окнах нужно только получить выделенный элемент или элементы на момент нажатия пользователем кнопки OK, Но иногда нужно реагировать на изменения в выделении элементов. Для этого используются два события:

События *ListBox* (выборочно)

Событие	Метод	Делегат	Параметр
SelectedIndexChanged	*OnSelectedIndexChanged*	*EventHandler*	*EventArgs*
SelectedValueChanged	*OnSelectedValueChanged*	*EventHandler*	*EventArgs*

Событие *SelectedValueChanged*, определенное в *ListControl*, доступно и в *ComboBox*. Эти два события в целом идентичны и происходят, когда программа или пользователь изменяют выделение. В списках *MultiSimple* эти события происходят, даже когда изменяется фокус, а не выделение,

Эта простая программа помещает в список имена переменных среды MS-DOS и показывает значение выделенного элемента,

EnvironmentVars.cs

```
//-----------------------------------------
// EnvironmentVars.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Collections;
using System.Drawing;
using System.Windows.Forms;

class EnvironmentVars: Form
 {
     Label label;

     public static void Main()
     {
          Application.Run(new EnvironmentVars());
```

см. след. < тр.

```
    }
    public EnvironmentVars()
    {
        Text = "Environment Variables";

            // Создаем метку.

        label = new Label();
        label.Parent   = this;
        label.Anchor   = AnchorStyles.Left | AnchorStyles.Right;
        label.Location = new Point(Font.Height, Font.Height);
        label.Size     = new Size(ClientSize.Width - 2 * Font.Height,
                                  Font.Height);

            // Создаем ListBox.

        ListBox listbox  = new ListBox();
        listbox.Parent   = this;
        listbox.Location = new Point(Font.Height, 3 * Font.Height);
        listbox.Size     = new Size(12 * Font.Height, 8 * Font.Height);
        listbox.Sorted   = true;
        listbox.SelectedIndexChanged +=
                    new EventHandler(ListBoxOnSelectedIndexChanged);

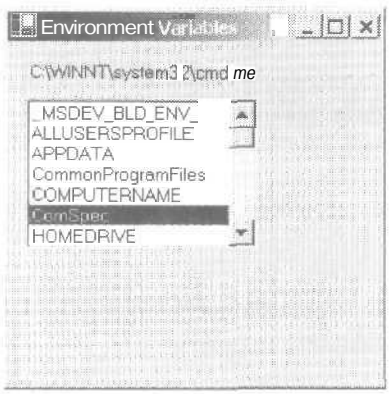
            // Помещаем в ListBox переменные среды.

        IDictionary dict = Environment.GetEnvironmentVariables();
        string[]    astr = new String[dict.Keys.Count];

        dict.Keys.CopyTo(astr, 0);
        listbox.Items.AddRange(astr);
        listbox.SelectedIndex = 0;
    }
    void ListBoxOnSelectedIndexChanged(object obj, EventArgs ea)
    {
        ListBox listbox = (ListBox) obj;
        string  strItem = (string) listbox.SelectedItem;

        label.Text = Environment.GetEnvironmentVariable(strItem);
    }
}
```

Заметьте: имена переменных среды находятся в массиве строк, добавляемом в список методом *AddRange*. В конце конструктора свойству ***SelectedIndex*** присваивается 0. При этом вызывается обработчик события *SelectedIndexChanged*, который определяет выделенный элемент и присваивает надписи значение переменной окружения для этого элемента. Вот пример работы программы:



Вы можете самостоятельно рисовать элементы списков. Для этого нужно присвоить значение свойству *DrawMode*. Значениями свойства являются члены перечисления *DrawMode*, определяющие, должны ли все элементы иметь одинаковую высоту или нет:

Перечисление *DrawMode*

Член	Значение
Normal	0
OwnerDrawFixed	1
OwnerDrawVariable	2

Кроме того, как обычно, нужно установить обработчики событий *MeasureItem* и *DrawItem*.

Окно списка + Текстовое поле = Комбинированное окно

В классическом варианте комбинированное окно выглядит как текстовое поле со стрелкой справа. Если ее щелкнуть, появится выпадающий список. Но этот традиционный способ использования комбинированных окон — не единственный. Вид комбинированного окна определяется свойством *DropDownStyle:*

Свойства *ComboBox* (выборочно)

Тип	Свойство	Доступ
ComboBoxStyle	*DropDownStyle*	Чтение/запись
boot	*DroppedDown*	Чтение/запись

Значениями свойства *DropDownStyle* являются члены перечисления *ComboBoxStyle*:

Перечисление *ComboBoxStyle*

Член	Значение	Описание
Simple	0	Поле можно редактировать, список всегда показывается
DropDown	1	Поле можно редактировать, список выпадает (по умолчанию)
DropDownList	2	Поле нельзя редактировать, список выпадает

То, что я называл *классическим* комбинированным окном, имеет стиль *DropDown*, применяемый по умолчанию. Пользователь может что-то ввести в поле редактирования или выбрать что-то из списка. Стиль *DropDownList* во многом напоминает обычный список за исключением того, что комбинированное окно такого стиля занимает меньше места, чем список. Свойство *DroppedDown* имеет значение *true*, если видима часть комбинированного окна, содержащая список.

Как и у окна списка, у комбинированного окна есть свойство *Items*, содержащее все элементы списка:

Свойства *ComboBox* (выборочно)

Тип	Свойство	Доступ
ComboBox.ObjectCollection	*Items*	Чтение

У класса *ObjectCollection* имеются свойство *Count*, индексатор и знакомые методы *Add*, *AddRange*, *Insert* и *Remove*.

Индекс выбранного элемента и сам выбранный элемент показывают свойства:

Свойства *ComboBox* (выборочно)

Тип	Свойство	Доступ
int	*SelectedIndex*	Чтение/запись
object	*SelectedItem*	Чтение/запись

У объекта *ListBox* свойства *SelectedIndex* и *SelectedItem* обычно имеют допустимые значения за исключением случая, когда список только что создан и эти свойства еще не инициализированы. Однако если пользователь в данный момент вводит что-либо в поле редактирования, свойство *SelectedIndex* объекта *ComboBox* равно −1, а *SelectedItem* — *null*.

Свойство *Text* всегда содержит текст, показываемый в поле редактирования. Так что свойство *Text* изменяется, когда пользователь выбирает разные элементы из списка и когда что-либо вводит в поле редактирования.

Ниже показаны события, используемые *ComboBox*:

События *ComboBox* (выборочно)

Событие	Метод	Делегат	Параметр
TextChanged	*OnTextChanged*	*EventHandler*	*EventArgs*
SelectedIndexChanged	*OnSelectedIndexChanged*	*EventHandler*	*EventArgs*
SelectionChangeCommitted	*OnSelectionChange Committed*	*EventHandler*	*EventArgs*

Событие *TextChanged* происходит, когда пользователь прокручивает элементы списка или вводит что-то в поле редактирования, событие *SelectedIndexChanged* — только когда пользователь прокручивает список. Для комбинированных окон стиля *DropDown* и *DropDownList*, если список раскрыт и пользователь прокручивает его элементы, происходят и событие *TextChanged*, и событие *SelectedIndexChanged*; событие *SelectionChangeCommitted* имеет место только при сворачивании списка. Однако если пользователь прокручивает элементы списка клавишами управления курсором, не раскрывая список, *SelectionChangeCommitted* происходит при каждом

изменении выделения. Для комбинированных окон *Simple* событие *SelectionChangedCommitted* происходит каждый раз при изменении выделения,

В усовершенствованной версии EnumerateEnumeration я заменил все три текстовых поля на комбинированные окна и для каждого из них установил обработчик события *TextChanged*. Обрабатывая события *TextChanged*, программа проверяет допустимость комбинации входных данных при нажатии клавиш в поле редактирования. Кнопку я удалил, так как она больше не нужна.

EnumerateEnumerationCombo.cs

```
//-------------------------------------------------------
// EnumerateEnumerationCombo.cs © 2001 by Charles Petzold
//-------------------------------------------------------
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Windows.Forms;

class EnumerateEnumerationCombo: Form
{
     CheckBox cbHex;
     ComboBox comboLibrary, comboNamespace, comboEnumeration;
     TextBox  tbOutput;

     const string strRegKeyBase =
         "Software\\ProgrammingWindowsWithCSharp\\EnumerateEnumerationCombo";

     public static void Main()
     {
          Application.Run(new EnumerateEnumerationCombo());
     }
     public EnumerateEnumerationCombo()
     {
          Text = "Enumerate Enumeration (Combo)";
          ClientSize - new Size(242, 164);

          Label label     - new Label();
          label.Parent    = this;
          label.Text      = "Library:";
          label.Location - new Point(8, 8);
          label.Size      = new Size(56, 8);

          comboLibrary               - new ComboBox();
          comboLibrary.Parent        = this;
          comboLibrary.DropDownStyle - ComboBoxStyle.DropDown;
          comboLibrary.Sorted        = true;
          comboLibrary.Location      = new Point(64, 8);
          comboLibrary.Size          - new Size(120, 12);
          comboLibrary.Anchor       | = AnchorStyles.Right;
          comboLibrary.TextChanged  +=
```

см. след. стр.

```
                new EventHandler(ComboBoxLibraryOnTextChanged);

label          - new Label();
label.Parent   = this;
label.Text     = "Namespace:";
label.Location = new Point(8, 24);
label.Size     = new Size(56, 8);

comboNamespace               - new ComboBox();
comboNamespace.Parent        = this;
comboNamespace.DropDownStyle = ComboBoxStyle.DropDown;
comboNamespace.Sorted        - true;
comboNamespace.Location      = new Point(64, 24);
comboNamespace.Size          - new Size(120, 12);
comboNamespace.Anchor        |= AnchorStyles.Right;
comboNamespace.TextChanged   +=
                new EventHandler(ComboBoxNamespaceOnTextChanged);

label          = new Label();
label.Parent   - this;
label.Text     = "Enumeration:";
label.Location = new Point(8, 40);
label.Size     = new Size(56, 8);

comboEnumeration               = new ComboBox();
comboEnumeration.Parent        * this;
comboEnumeration.DropDownStyle = ComboBoxStyle.DropDown;
comboEnumeration.Sorted        = true;
comboEnumeration.Location      = new Point(64, 40);
comboEnumeration.Size          = new Size(120, 12);
comboEnumeration.Anchor        |= AnchorStyles.Right;
comboEnumeration.TextChanged   +=
                new EventHandler(ComboBoxEnumerationOnTextChanged);

cbHex                = new CheckBox();
cbHex.Parent         = this;
cbHex.Text           = "Hex";
cbHex.Location       = new Point(192, 25);
cbHex.Size           = new Size(40, 8);
cbHex.Anchor         = AnchorStyles.Top | AnchorStyles.Right;
cbHex.CheckedChanged += new EventHandler(CheckBoxOnCheckedChanged);

tbOutput            = new TextBox();
tbOutput.Parent     = this;
tbOutput.ReadOnly   = true;
tbOutput.Multiline  = true;
tbOutput.ScrollBars - ScrollBars.Vertical;
tbOutput.Location   = new Point(8, 56);
tbOutput.Size       = new Size(226, 100);
tbOutput.Anchor     = AnchorStyles.Left | AnchorStyles.Top |
```

он. след. стр.

```
                        AnchorStyles.Right | AnchorStyles.Bottom;

    AutoScaleBaseSize = new Size(4, 8);

        // Инициализировать отображение.

    FillComboBox(comboLibrary, strRegKeyBase);
    UpdateTextBox();
J

void ComboBoxLibraryOnTextChanged(object obj, EventArgs ea)
{
    FillComboBox(comboNamespace, strRegKeyBase + "\\" +
                                 comboLibrary.Text);

    ComboBoxNamespaceOnTextChanged(obj, ea);
}
void ComboBoxNamespaceOnTextChanged(object obj, EventArgs ea)
{
    FillComboBox(comboEnumeration, strRegKeyBase + "\\" +
                                   comboLibrary.Text + "\\" +
                                       comboNamespace.Text);

    ComboBoxEnumerationOnTextChanged(obj, ea);
}
void ComboBoxEnumerationOnTextChanged(object obj, EventArgs ea)
{
    UpdateTextBox();
}
void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
{
    UpdateTextBoxO;
}
void UpdateTextBox()
{
    if (EnumerateEnumeration.FillTextBox(tbOutput, comboLibrary.Text,
        comboNamespace.Text, comboEnumeration.Text, cbHex.Checked))
    {
        if (!comboLibrary.Items.Contains(comboLibrary.Text))
            comboLibrary.Items.Add(comboLibrary.Text);

        if (!comboNamespace.Items.Contains(comboNamespace.Text))
            comboNamespace.Items.Add(comboNamespace.Text);

        if (!comboEnumeration.Items.Contains(comboEnumeration.Text))
            comboEnumeration.Items.Add(comboEnumeration.Text);

        string strRegKey = strRegKeyBase + "\\" +
                                   comboLibrary.Text + "\\" *
                                       comboNamespace.Text * "\\" +
```

см. след. стр.

```
                                                    comboEnumeration.Text;
            RegistryKey regkey =
                              Registry.CurrentUser.OpenSubKey(strRegKey);

            if (regkey == null)
                regkey = Registry.CurrentUser.CreateSubKey(strRegKey);

            regkey.Close();
        }
    }

    bool FillComboBox(ComboBox combo, string strRegKey)
    {
        combo.Items.Clear();

        RegistryKey regkey = Registry.CurrentUser.OpenSubKey(strRegKey);

        if (regkey 1» null)
        {
            stringE] astrSubKeys - regkey.GetSubKeyNames();
            regkey.Close();

            if (astrSubKeys.Length > 0)
            {
                combo.Items.AddRange(astrSubKeys);
                combo.SelectedIndex = 0;
                return true;
            }
        }
        combo.Text - "";
        return false;
    }
}
```

При первом запуске программы все три комбинированных окна пусты. На пользователя возлагается ответственность по вводу правильных имени библиотеки, пространства имен и имени перечисления. Как я уже говорил, программа проверяет допустимость комбинации входных данных при каждом нажатии клавиши. Если комбинация допустима, информация о перечислении показывается статическим методом *FillTextBox* исходной программы EnumerateEnumeration.

Кроме того, программа сохраняет допустимые комбинации в реестре, и каждая допустимая комбинация затем добавляется в комбинированные окна. Древовидная структура реестра отлично подходит для применения в таких приложениях. Запустив редактор реестра после ввода нескольких допустимых комбинаций библиотеки, пространства имен и перечисления, вы увидите, что в реестре нет «настоящих» данных! Каждая допустимая комбинация становится разделом, Вы увидите записи с такими разделами:

```
Software\ProgrammingWindowsWithCSharp\EnumerateEnumerationCombo
 \system.drawing\System.Drawing.Drawing2D\DashStyle
```

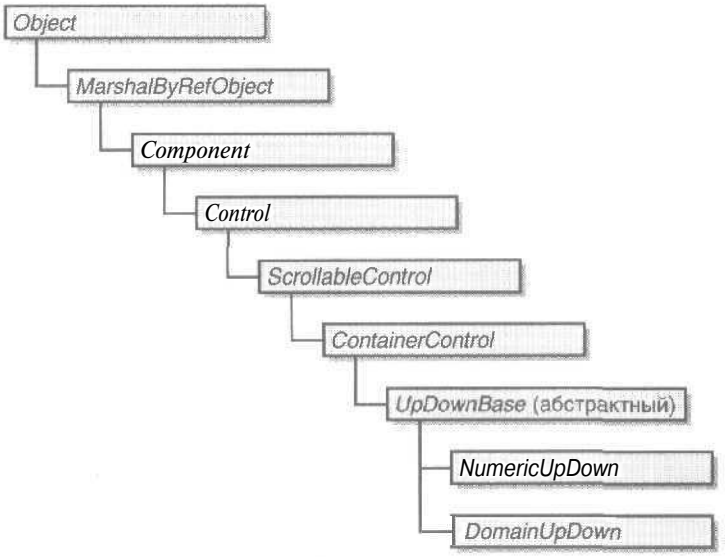
или такими:

```
Software\ProgrammingWindowsWithCSharp\EnumerateEnumerationCombo
 \system.windows.forms\System.Windows.Forms\DockStyle
```

При изменении выделения или текста в первом комбинированном окне (в которое вводится имя библиотеки) метод *FillComboBox* получает список подразделов реестра для данного имени библиотеки. Аналогично при изменении выделения или текста комбинированного окна для ввода пространства имен *FillComboBox* получает список допустимых перечислений для третьего комбинированного окна. Чем чаще запускать эту программу, тем полезнее она становится.

Управляющие стрелки

Управляющие стрелки (up-down control) Windows Forms традиционно известны как *счетчик с прокруткой* (spin control). Как видите, имеется два типа управляющих стрелок:



Управляющие стрелки состоят из поля редактирования и пары стрелок справа. Элемент управления *NumericUpDown* позволяет выбирать числа из диапазона, тогда как *DomainUpDown* позволяет выбирать из рабора объекты, которые идентифицируются строками. Б этом разделе я сосредоточу внимание на *NumericUpDown*.

Обычно для инициализации этого элемента задаются значения таким свойствам:

Свойства *NumericUpDown* (выборочно)

Тип	Значение	Доступ
decimal	*Value*	Чтение/запись
decimal	*Minimum*	Чтение/запись
decimal	*Maximum*	Чтение/запись
decimal	*Increment*	Чтение/запись

Обратите внимание на использование типа *decimal* (подробнее о *decimal* см. приложение Б). Свойство *Increment* показывает, на сколько изменяется *Value* при щелчке «стрелки-вверх» или «стрелки-вниз». По умолчанию *Value* может находиться в диапазоне от 0 до 100, а *Increment* равно 1. Можно снять практически все ограничения, присвоив свойствам *Minimum* и *Maximum* значения *Decimal.MinValue* и *Decimal.MaxValue*.

Свойства *Minimum* и *Maximum* очень строгие: если пользователь вручную введет число вне диапазона от *Minimum* до *Maximum,* элемент управления изменит число или на *Minimum,* или на *Maximum.* (Осторожнее с этим при создании модальных диалоговых окон! Может случиться, что пользователь, нажав кнопку OK, закроет диалоговое окно и не заметит, что значение изменилось.) Если значение, лежащее вне диапазона, присваивается свойству *Value* программно, генерируется исключение.

Следующие свойства управляют показом чисел в счетчике:

Свойства *NumericUpDown* (выборочно)

Тип	Свойство	Доступ	Значение по умолчанию
int	*DecimalPlaces*	Чтение/запись	0
bool	*ThousandsSeparator*	Чтение/запись	*false*
bool	*Hexadecimal*	Чтение/запись	*false*

Событие *ValueChanged* происходит, когда программа или пользователь изменяют значение счетчика:

Свойства *NumericUpDown* (выборочно)

Событие	Метод	Делегат	Параметр
ValueChanged	*OnValueChanged*	*EventHandler*	*EventArgs*

Для демонстрации работы с элементами управления *NumericUpDown* я написал программу, использующую 9 управляющих стрелок. Программа Transform позволяет экспериментировать с преобразованиями матрицы путем изменения *6* элементов матрицы или вызова различных методов класса *Matrix,* например *Scale* или *Shear.* Я разделил программу на три файла: один содержит форму, два других - два диалоговых окна.

Большая часть главной формы состоит из двух методов, вызываемых при выполнении события *Paint: DrawAxes* рисует координатную систему, а *DrawHouse* — домик. Домик рисуется с использованием объекта *Matrix,* который хранится как поле.

Transform.cs

```
//-------------------------------------
// Transform.cs $ 2001 by Charles Petzold
//-------- , -------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;      // Для растровых изображений
```

см. след. стр.

```
using System.Windows.Forms;

class Transform: For»
{
    Matrix matrix = new Matrix();

    public static void Main()
    {
        Application.Run(new Transform());
    }
    public TransformO
    {
        Text = "Transform";
        ReslzeRedraw = true;
        BackColor * Color.White;
        Size += Size;

            // Создаем модальное диалоговое окно.

        MatrixElements dlg = new MatrixElefnentsO;
        dlg.Owner    = this;
        dlg.Matrix   * matrix;
        dlg.Changed += new EventHandler(MatrixDialogOnChanged);
        dlg.Show();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx - pea.Graphics;

        DrawAxes(grfx);
        grfx.Transform = matrix;
        DrawHouse(grfx);
    }
     void DrawAxes(Graphics grfx)
    {
        Brush        brush  = Brushes.Black;
        Pen          pen    = Pens.Black;
        StringFormat strfmt = new StringFormat();

            // Горизонтальная ось

        strfmt.Alignment = StringAlignment.Center;

        for (int i = 1; i <= 10; i++)
        {
            grfx.DrawLine(pen, 100 * i, 0, 100 * i, 10);
            grfx.DrawString((i * 100).ToString(), Font, brush,
                            100 * i, 10, strfmt);
            grfx.DrawLine(pen, 100 * i, 10 + Font.Height, •'  ,
                               100 * 1, ClientSize.Height);
```

см. след. стр.

```
        }

            // Вертикальная ось

        strfmt.Alignment     = StringAlignment.Near;
        strfmt.LineAlignment - StringAlignment.Center;

        for (int 1 = t; i <= 10; i++)
        {
            grfx.DrawLine(pen, 0, 100 * i, 10, 100 * i);
            grfx.DrawString((i * 100).ToString(), Font, brush,
                            10, 100 * i, strfmt);
            float cxText = grfx.MeasureString(
                                (i * 100).ToString(), Font).Width;
            grfx.DrawLine(pen, 10 + cxText,      100 * i,
                          ClientSize.Width, 100 * i);
        }
    }
    void DrawHouse(Graphics grfx)
    {
        Rectangle   rectfacade  =  new RectangleC 0, 40, 100,  60);
        Rectangle   rectDoor    =  new Rectangle(10, 50,  25,  50);
        Rectangle[] rectWindows - { new Rectangle(50, 50,  10,  10),
                                    new Rectangle(60, 50,  10,  10),
                                    new Rectangle{70, 50,  10,  10),
                                    new Rectangle(50, 60,  10,  10),
                                    new Rectangle(60, 60,  10,  10),
                                    new Rectangle(70,60,   10,  10),
                                    new Rectangle(15, 60,   5,   7),
                                    new Rectangle(20, 60,   5,   7),
                                    new Rectangle(25, 60,   5,   7) };
        Rectangle   rectChimney =  new Rectangle(80,  5,  10,  35);
        Point[]     ptRoof      = { new Point( 50,  0),
                                    new Point(  0, 40),
                                    new Point(100, 40) };

            // Создание растрового изображения и кисть для трубы.

        Bitmap bitmap = new Bitmap(8, 6);

        byte[] bits = { 0, 0, 0, 0, 0, 0, 0, 0,
                        1, 1, 1, 0, 1, 1, 1, 0,
                        1, 1, 1, 0, 1, 1, 1, 0,
                        0, 0, 0, 0, 0, 0, 0, 0,
                        1, 0, 1, 1, 1, 0, 1, 1,
                        1, 0, 1, 1, 1, Q, 1, 1,

                        0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 1, 1, 1, 1, 1, 1 };
```

см. след. стр.

```
        for (int i = 0; i < 48; i++)
            bitmap.SetPixel(i % 8, i / 8,
                            bits[i] == 1 ? Color.DarkGray: Color.LightGray);

        Brush brash = new TextureBrush(bitmap);

        // Рисуем весь дом.

        grfx.FillRectangle(Brushes.LightGray, rectFacade);
        grfx.DrawRectangle(Pens.Black,        rectFacade);

        grfx.FillRectangle(Brushes.DarkGray, rectDoor);
        grfx.DrawRectangle(Pens.Black,       rectDoor);

        grfx.FillRectangles(Brushes.White, rectWindows);
        grfx.DrawRectangles(Pens.Black,    rectWindows);

        grfx.FillRectangle(brush,      rectChimney);
        grfx.DrawRectangle(Pens.Black, rectChimney);

        grfx.FillPolygon(Brushes.DarkGray, ptRoof);
        grfx.DrawPolygon(Pens.Black,       ptRoof);
    }
    void MatrixDialogOnChanged(object obj, EventArgs ea)
    {
        MatrixElements dlg = (MatrixElements) obj;
        matrix = dlg.Matrix;
        Invalidate();
    }
}
```

Кроме того, программа показывает немодальное диалоговое окно Matrix Elements, реализованное в показанном ниже исходном файле. Диалоговое окно содержит 6 элементов управления *NumericUpDown*, позволяющих ввести 6 элементов матрицы, и обрабатывает событие *Changed*. При щелчке кнопки происходит событие *Changed*, сигнализирующее основной программе об изменении матрицы. (Основная программа обрабатывает это событие в методе *MatrixDialogOnChanged.)* Объект *Matrix* доступен как свойство.

MatrixEtements.cs

```
//-------------------------------------------
// MatrixElements.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class MatrixElements: Form
```

см. след. стр.

```
{                                          i
    Matrix          fflatrix;
    Button          btnUpdate;
    NumericUpDown[] updown = flew NumericUpDown[6];

    public event EventHandler Changed;

    public MatrixElements()
    {
        Text * "Matrix Elements";
        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox      = false;
        MinimizeBox     = false;
        MaximizeBox     * false;
        ShowInTaskbar   = false;

        String[] strLabel = < "X Scale:",     "Y Shear:",
                              "X Shear:",     "Y Scale:",
                              "X Translate:", "Y Translate:" };

        far (int 1 * S; i < 6; i++)
        {
            Label label    = new Label();
            label.Parent   = this;
            label.Text     * strLabel[i];
            label.Location = new Point(8, 8 + 16 * i);
            label.Size     = new Size(64, 8);

            updown[i] * new NumericUpDown();
            updown[i].Parent        * this;
            updownti].Location     = new Point(76, 8 + 18 * i);
            updown[i].Size          = new Size(48, 12);
            updown[i].TextAlign     = HorizontalAlignment.Right;
            updown[i].ValueChanged += new EventHandler
                                          (UpDownOnValueChanged);
            updown[i].DecimalPlaces = 2;
            updown[i].Increment * 0.1m;
            updown[i].Minimum = Decimal.MinValue;
            updown[i].Maximum = Decimal.MaxValue;
        }
        btnUpdate          * new Button();
        btnUpdate.Parent   = this;
        btnUpdate.Text     = "Update";
        btnUpdate.Location = new Point(8, 108);
        btnUpdate.Size     * new Size (50, 16);
        btnUpdate.Click   += new EventHandler(ButtonUpdateOnClick);

        AcceptButton = btnUpdate;

        Button btn .. = new Button();
```

см. след. стр.

```
        btn.Parent   = this;
        btn.Text     = "Methods...";
        btn.Location * new Point(76, 108);
        btn.Size     = new Size(50, 16);
        btn.Click   += new EventHandler(ButtonMethodsOnClick);

        ClientSize = new Size(134, 132);

        AutoScaleBaseSize - new Size(4, 8);
    }
    public Matrix Matrix
    {
        get
        {
            matrix = new Matrix((float) updown[0].Value,
                                (float) updown[1].Value,
                                (float) updown[2].Value,
                                (float) updown[3].Value,
                                (float) updown[4].Value,
                                (float) updown[5].Value);
            return matrix;
        }
        set
        {
            matrix = value;

            for (int i = 0; i < 6; i++)
                updown[i].Value = (decimal) value.Elements[i];
        }
    }
    void UpDownOnValueChanged(object obj, EventArgs ea)
    {
        Graphics grfx = CreateGraphics();

        bool boolEnableButton = true;

        try
        {
            grfx.Transform = Matrix;
        }
        catch
        {
            boolEnableButton = false;
        }

        btnUpdate.Enabled = boolEnableButton;
        grfx.Dispose();
    }
    void ButtonUpdateOnClick(object obj, EventArgs ea)
    {
```

см. след. стр.

```
            if (Changed != null)
                Changed(this, new EventArgs());
        }
        void ButtonMethodsOnClick(object obj, EventArgs ea)
        {
            MatrixMethods dlg = new MatrixMethods();

            dlg.Matrix = Matrix;

            if (dlg.ShowDialog() == DialogResult.OK)
            {
                Matrix *= dlg.Matrix;
                btnUpdate.PerformClick();
            }
        }
}
```

Диалоговое окно Matrix Elements содержит еще одну кнопку — Methods, которая вызывает модальное диалоговое окно Matrix Methods, реализованное в следующем исходном файле. Matrix Methods содержит еще 3 элемента управления *NumericUpDown*, в которые вводятся параметры методов класса *Matrix*, изменяющих элементы матрицы. Кнопки, по которым вызываются методы, еще и закрывают диалоговое окно.

MatrixMethods.cs

```
//-----------------------------------------
// MatrixMethods.cs © 2001 by CHarles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class MatrixMethods: Form
{
    Matrix          matrix;
    Button          btnInvert;
    NumericUpDown[] updown = new NumericUpDown[3];
    RadioButton[]   radio = new RadioButton[2];

    public MatrixMethods()
    {
        Text *= "Matrix Methods";
        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox      = false;
        MinimizeBox     = false;
        MaximizeBox     *= false;
        ShowInTaskbar   = false;
        Location .      = ActiveForm.Location +
```

См. след. стр.

```
                SystemInformation.CaptionButtonSize + ,,
                SystemInformation.FrameBorderSize;

String[] astrLabel = { "X / DX:", "Y / DY:", "Angle:" };

for(inti=0; i<3;i++)
{
     Label label    = new Label();
     label.Parent   = this;
     label.Text     = astrLabel[i];
     label.Location * new Point(8, 8 + 16 * i);
     label.Size     * new Size(32, 8);

     updown[i] = new NumericUpDown();
     updown[i].Parent        = this;
     updown[i].Location      = new Point(40, 8 * 16 * i);
     updown[i].Size          = new Size(48, 12);
     updown[i].TextAlign     = HorizontalAlignment.Right;

     updown[i].DecimalPlaces - 2;
     updown[i].Increment = 0.1m;
     updown[i].Minimum = Decimal.MinValue;
     updown[i].Maximum = Decimal.MaxValue;
}
     // Создаем группу и переключатели.

GroupBox grpbox = new GroupBox();
grpbox.Parent   - this;
grpbox.Text     - "Order";
grpbox.Location = new Point(8, 60);
grpbox.Size     = new Size(80, 32);

for (int i = 0; i < 2; i++)
{
     radio[i] = new RadioButton();
     radio[i].Parent - grpbox;
     radio[i].Text = new string[] { "Prepend", "Append" } [i];
     radio[i].Location = new Point(8, 8 + 12 * i);
     radio[i].Size     - new Size(50, 10);
     radio[i].Checked = <1 == 0);
}

     // Создаем восемь кнопок, закрывающих диалоговое окно.

string[] astrButton - { "Reset", "Invert", "Translate", "Scale",
                        "Rotate", "RotateAt", "Shear", "Cancel" };

EventHandler[] aeh = { new EventHandler(ButtonResetOnClick),
                       new EventHandler(ButtonInvertOnClick),
                       new EventHandler(ButtonTranslateOnClick),
```

см. след. стр.

```
                                new EventHandler(ButtonScaleOnClick),
                                new EventHandler(ButtonRotateOnClick),
                                new EventHandler(ButtonRotateAtOnClick),
                                new EventHandler(ButtonShearOnClick) };

        for (int 1 = 0; i < 8; i++)
        {
            Button btn = new Button();
            btn.Parent = this;
            btn.Text   = astrButton[i];
            btn.Location = new Point(100 + 72 * (i > 3 ? 1 : 0),
                                     8 * (i X 4) * 24);
            btn.Size     = new Size(64, 14);

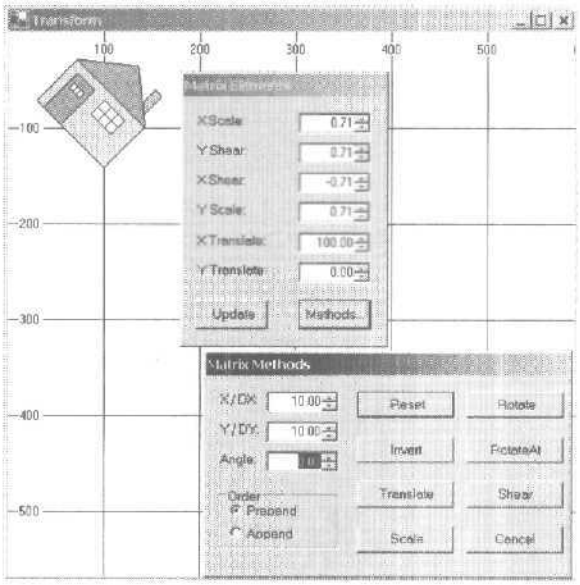
            if <i == 0)    // Кнопка сброса
            {
                AcceptButton * btn;
            }
            if (i == 1)    // Кнопка инвертирования
            {
                btnInvert * btn;
            }
            if (i < 7)     // Все кнопки, кроме Cancel
            {
                btn. Click += aeh[i];
                btn.DialogResult = DialogResult.OK;
            }
            else           // Кнопка Cancel
            {
                btn.DialogResult = DialogResult.Cancel;
                CancelButton = btn;
            }
        }
        ClientSize = new Size(240, 106);

        AutoScaleBaseSize = new Size(4, 8);
    }
    public Matrix Matrix
    {
        get
        {
            return matrix;
        }
        set ·
        {
            matrix = value;
            btnInvert.Enabled = matrix. IsZnvertible;
        }
    }
    void ButtonResetOnClick(object obj, EventArgs ea)
```

см.след.стр.

```
    {
        matrix.flesetO;
    }
    void ButtonInvertOnClick(object obj, EventArgs ea)
    {
        matrix.Invert();
    }
    void ButtonTranslateOnClick(object obj, EventArgs ea)
    {
        matrix.Translate((float) updown[0].Value,
                         (float) updown[1].Value,
            radio[0].Checked ? MatrixOrder.Prepend : MatrixOrder.Append);
    }
    void ButtonScaleOnClick(object obj, EventArgs ea)
    {
        matrix.Scale((float) updown[0].Value,
                     (float) updown[1].Value,
            radio[0].Checked ? MatrixOrder.Prepend : MatrixOrder.Append);
    }
    void ButtonRotateOnClick(object obj, EventArgs ea)
    {
        matrix.ftotate((float) updown[2].Value,
            radio[0].Checked ? MatrixOrder.Prepend : MatrixOrder.Append);
    }
    void ButtonRotateAtOnClick(object obj, EventArgs ea)
    {
        fflatrix.ftotateAt« float) updown[2].Value,
                        new PointF((float) updown[0].Value,
                                   (float) updown[1].Value),
            radio[0].Checked ? MatrixOrder.Prepend : MatrixOrder.Append);
    }
    void ButtonShearOnClick(object obj, EventArgs ea)
    {
        matrix.Shear((float) updown[0].Value,
                     (float) updown[1].Value,
            radio[0].Checked ? MatrixOrder.Prepend : MatrixOrder.Append);
    }
}
```

Вот как выглядит программа, когда показаны оба диалоговых окна:



Труба домика нарисована кистью *TextureBrush*, которая использует изображение *Bitmap,* выглядящее как кирпичи. Я мог бы применить и *HatchBrush* со стилем *HatchStyleHorizontalBrick*. Но, как вы помните по главе 11, к *HatchBrush* нельзя применять преобразования, а к *TextureBrush* — можно. Если сделать дом больше, то и кирпичи в трубе станут больше. Можно также сдвигать и вращать кирпичи. Заметьте: в GDI+ применяется алгоритм сглаживания, позволяющий избежать эффекта огромных пикселов.

ГЛАВА

19



Игры со шрифтами

Шрифты TrueType и OpenType, доступные программам Windows Forms, являются контурными. Это значит, что каждый знак определяется набором прямых линий и сплайнов. Как мы видели в главе *9* и последующих, такие шрифты можно плавно масштабировать. Определения шрифтов содержат разметку, которая помогает избежать искажений, возникающих при преобразовании координат с плавающей запятой в дискретные размеры координатной сетки,

Так как знаки шрифта определяются набором прямых линий и кривых, они хорошо интегрируются с остальной частью графической системы Windows Forms. Знаки шрифта можно преобразовать, окрасить любыми кистями и сделать частью графического контура,

Применение к тексту этих технологий программирования графики (как по отдельности, так и в различных сочетаниях) приводит к тому, что текст перестает быть просто объектом для чтения и становится чем-то вроде искусства.

Приступим к делу

Начиная с главы 5, мы используем классы, производные от *PrintableForm,* для демонстрации графических технологий на мониторе и на принтере. В классах, унаследованных от *PrintableForm,* переопределяется метод *DoPage,* применяемый для вывода графики. *DoPage* вызывается в методе *OnPaint* формы, чтобы прорисовать клиентскую область. Когда вы щелкаете клиентскую область, вновь вызывается метод *DoPage* для вывода графики на принтер.

В этой главе я хочу определить новый класс — *FontMenuForm,* который наследуется от *PrintableForm* и содержит единственный пункт меню «Font!». Это меню вызывает стандартное диалоговое окно Font (Шрифт), позволяющее изменить п< >ле *font.* Класс *FontMenuForm* также содержит пару методов, удобных для вывода текста.

FontMenuForm.cs

```
// ----- ..... ----------
// FontMenuForm.cs © 2001 by Charles Petzold
//                 —
using System;
using System.Drawing;
using System.Windows.Forms;

class FontMenuForm: PrintableForm
{
     protected string strText = "Sample Text";
     protected Font font - new Font("Times Newfloman",24, FontStyle.Italic);

     public new static void Main()
     {
          Application.Run(new FontMenuForm());
     }
     public  FontMenuForm()
     {
          Text = "Font Menu Form";
          Menu = new MainMenu();
          Menu.MenuItems.Add("&Font!", new EventHandler(MenuFontOnClick));
     }
     void MenuFontOnClick(object obj, EventArgs ea)
     {
          FontDialog dlg - new FontDialog();
          dlg.Font * font;

          if (dlg.ShowDialog() == DialogResult.OK)
          {
               font = dlg.Font;
               Invalidate();
          1
     }
     protected override void DoPage(Graphics flrfx, Color clr, int cx, int cy)
     {
          SizeF sizef = grfx.HeasureString(strText, font);
          Brush brush = new SolidBrush(clr);

          grfx.DrawString(strText, font, brush, (cx - sizef.Width) / 2,
                                                (cy - sizef.Height) / 2);
     }
     public float Get Ascent( Graphics grfx, Font font)
     {
          return font.GetHeight(grfx) *
                    font.FontFamily.GetCellAscent(font.Style) /
                         font.FontFamily.GetLineSpacing(font.Style);
     }
     public float GetDescent(Graphics fl'"^5*. Font font)
```

см. след. стр.

```
    {
        return font.GetHeight(grfx) *
                    font.FontFamily.GetCellDescent(font.Style) /
                        font.FontFamily.GetLineSpacing(font.Style);
    }
    public float PointsToPageUnits(Graphics grfx, Font font)
    {
        float fFontSize;

        if (grfx.PageUnit == GraphicsUnit.Display)
            fFontSize = 100 * font.SizeInPoints / 72;
        else
            fFontSize = grfx.DpiX * font.SizeInPoints / 72;

        return fFontSize;
    }
}
```

Программа, производная от класса *FontMenuForm*, должна переопределить метод *DoPage* (как и в случае наследования от класса *PrintableForm*), а также задействовать поля *font* и *strText*. Эти поля инициализируются в конструкторе класса различными начальными значениями.

Методы *GetAscent* и *GetDescent* вычисляют высоту с надстрочным элементом и высоту подстрочного элемента для определенного шрифта, используя технологию, рассмотренную в главе 9. Метод *PointToPageUnit* вычисляет размер шрифта в единицах страницы. Метод предполагает, что для объекта *Graphics*, передаваемого в качестве аргумента, указываются единицы измерения страницы по умолчанию. Для принтеров единицами измерения страницы по умолчанию являются *GraphicsUnit.Display*, а для мониторов — *GraphicsUnit.Pixel*.

Закрашивание текста кистью

В главе 17 я не показал, как использовать все разнообразие доступных кистей при работе с текстом — я приберег это для этой главы. Вот программа, которая выводит текст, используя *HatchBrush*, созданный при помощи *HatchStyle.HorizontalBrick*.

Bricks.cs

```
//---------------------------------------------
// Bricks.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class Bricks: FontMenuForm
{
    public new static void Main()
```

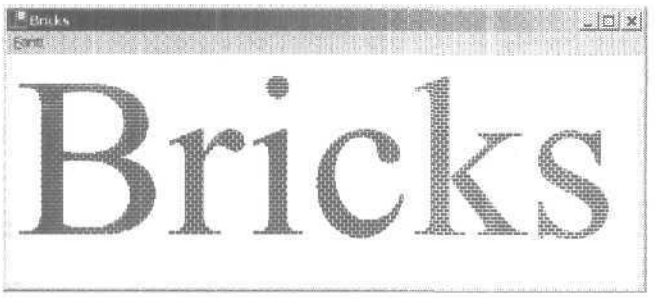
см. след. а пр.

```
    {
        Application.Run(new Bricks());
    }
    public Bricks()
    {
        text = "Bricks";

        strText = "Bricks";
        font = new Font("Times New Roman", 144);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        SizeF sizef = grfx.MeasureString(strText, font);
        Brush hbrush = new HatchBrush(HatchStyle.HorizontalBrick,
                                      Color.White, Color.Black);

        grfx.DrawString(strText, font, hbrush, (cx - sizef.Width) / 2,
                                               (cy - sizef.Height) / 2);
    }
}
```

Если увеличить клиентскую область, вывод будет выглядеть так:



Штриховые кисти работают лучше с большими шрифтами. С помощью узких штриховых кистей внешний вид можно улучшить, используя оконтурнвание знаков. Я продемонстрирую эту методику ниже.

Следующая программа выводит текст, созданный с помощью градиентной кисти.

GradientText.cs

```
//-------------------------------------------
// GradientText.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class GradientText: FontMenuForm
{
```

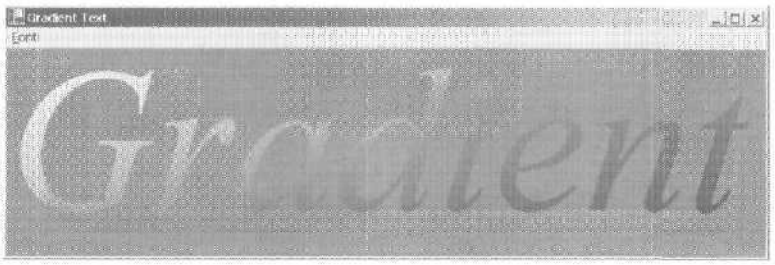
см. след. стр.

```
    public new static void Main()
    {
        Application.Run(new GradientText());
    }
    public GradientText()
    {
        Text = "Gradient Text";
        Width *= 3;
        strText = "Gradient";
        font = new Font("Times New Roman", 144, FontStyle.Italic);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        SizeF  sizef  * grfx.MeasureString(strText, font);
        PointF ptf    = new PointF((cx - sizef.Width) / 2,
                                   (cy - sizef.Height) / 2);

        RectangleF rectf - new flectangleF(ptf, sizef);

        LinearGradientBrush lgbrush = new LinearGradientBrush(rectf,
                                      Color.White, Color.Black,
                                      LinearGradientMode.ForwardDiagonal);
        grfx.Clear(Color.Gray);
        grfx.DrawString(strText, font, lgbrush, ptf);
    }
}
```

Текст выполнен в белом цвете в левом верхнем углу и в черном цвете — в правом нижнем, на сером фоне:



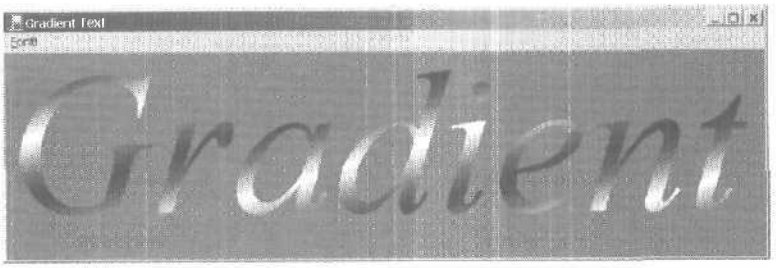
Если вставить две строчки;

```
sizef.Width /= 8;
sizef.Height /= 8;
```

перед *RectangleF* и строку:

```
lgbrush.WrapMode = WrapMode.TileFlipXY;
```

после создания кисти вы получите небольшую мозаичную кисть, которая выглядит так:



Можно достичь интересных эффектов, используя простую сплошную кисть, как показано в программе DropShadow.

DropShadow.cs

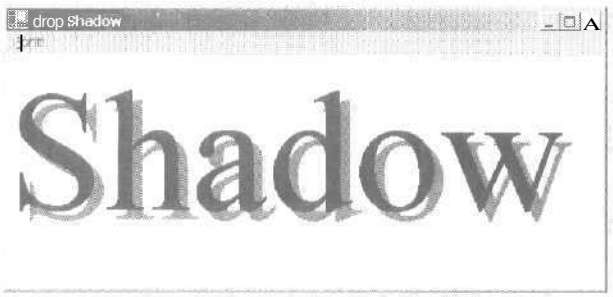
```
//---------------------------------------
// propShadow.cs @ 2001 by Charles Petzold
//---- ___,_________--------_____ -------
using System;
using System.Drawing;
using System.Windows.Forms;

class DropShadow: FontMenuForm
{
     const int iOffset = 10;  // Примерно (для принтера точно) 1/10 дюйма.

     public new static void Main()
     {
          Application.Run(new DropShadow());
     }
     public DropShadow()
     {
          Text = "Drop Shadow";
          Width *= 2;
          strText = "Shadow";
          font = new Font("Times New Roman", 108);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          SizeF sizef = grfx.MeasureString(strText, font);
          float x     = (cx - sizef.Width) / 2;
          float y     * (cy - sizef.Height) / 2;

          grfx.Clear(Color.White);
          grfx.DrawString(strText, font, Brushes.Gray, x, y);
          grfx.DrawString(strText, font, Brushes.Black, x - iOffset,
                                                        y - iOffset);
     }
}
```

Эта программа создает эффект падающей тени, рисуя текст двумя разными кистями, разнесенными на 10 единиц:



При очень маленьком разносе и правильном выборе цветов можно достичь рельефного или гравировочного эффекта:

EmbossedText.cs

```
//-------------------------------------
// EmbossedText.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class EmbossedText: FontMenuForm
{
     int iOffset = 2;

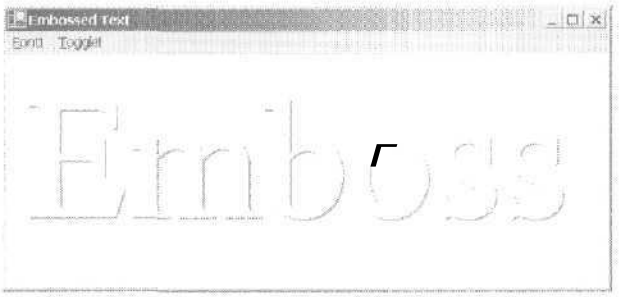
     public new static void Main()
     {
          Application.Run(newEmbossedText());
     }
     public EmbossedText()
     {
          Text = "Embossed Text";
          Width *= 2;
          Menu.MenuItems.Add("&Toggle!",
                                  new EventHandler(MenuToggleOnClick));
          strText = "Emboss";
          font = new Font("Times New Roman", 108);
     }
     void MenuToggleOnClick(object obj, EventArgs ea)
     {
          iOffset = -iOffset;
          Text = (iOffset > 0) ? "Embossed Text" : "Engraved Text";
          strText = (iOffset > 0) ? "Emboss" : "Engrave";
          Invalidate();
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          SizeF sizef = grfx.MeasureString(strText, font);
          float x     = (cx - sizef.Width) / 2;
```

см. след. стр.

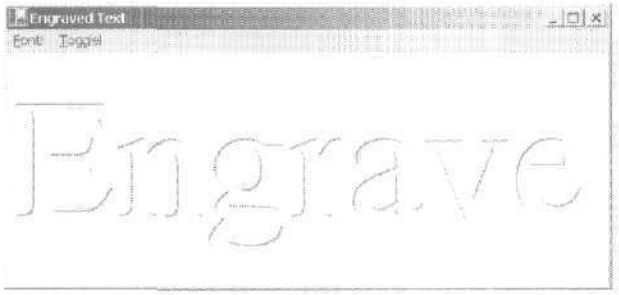
```
            float y      * (cy - sizef.Height) / 2;

            grfx.Clear(Color.White);
            grfx.DrawString(strText, font, Brushes.Gray, x, y);
            grfx.DrawString(strText, font, Brushes.White, x - iOffset,
                                                          y - iOffset);
        }
    }
```

Эта программа рисует серый текст, а затем белый текст на белом фоне:



EmbossedText содержит меню «Toggle!», позволяющее переключиться на гравировочный эффект:



По сути эти два эффекта одинаковы. Единственная разница — в выборе положительного или отрицательного сдвига между двумя выводимыми текстами. Так как мы привыкли, что источник света находится сверху, появление тени внизу и справа от знаков интерпретируется как выступающий текст, а тень сверху и слева — как утопленный. Перевернув эту книгу (или ваш монитор) вверх ногами, вы увидите, что эффект изменится на противоположный.

Как показано в программе BlockFont, если рисовать одну и ту же текстовую строку несколько раз в одном и том же цвете, можно достичь объемного эффекта.

BlockFont.cs

```
//-------------------------------------
// SlockFont.es © 2001 by Charles Petzold
//-------------------------------------
using System;
```

см. след. стр.

```
using System.Drawing;
using System.Windows.Forms;

class BlockFont: FontMenuForm
{
     const int iReps = 50;    // Примерно (для принтера точно) 1/2 дюйма.

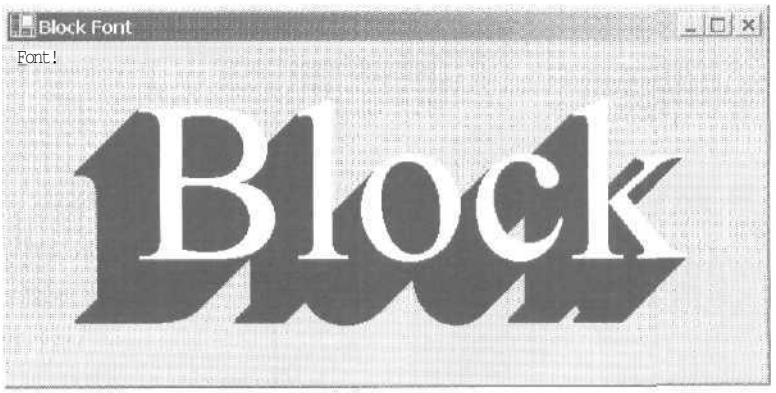
     public new static void Main()
     {
          Application. Run(new BlockFont0);
     }
     public BlockFont()
     {
          Text - "Block Font";
          Width *= 2;
          strText = "Block";
          font * new Font("Times New Roman", 108);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          SizeF sizef = grfx.MeasureString(strText, font);
          float x     = (cx - sizef.Width - iReps) / 2;
          float y     = (cy - sizef.Height + iReps) / 2;

          grfx.Clear(Color.LightGray);

          for (int i = 0; i < iReps; i++)
               grfx.DrawString(strText, font, Brushes.Black, x + i, y - i);

          grfx.DrawString(strText, font, Brushes.White, x + iReps,
                                                        y - iReps);
     }
}
```

После рисования нескольких строк черным цветом метод *DoPage* рисует последнюю строку белым. Вот результат:



Возможно, сверху вы захотите поместить оконтуренный шрифт, это даст более выразительный вид на белом фоне.

Преобразование шрифта

Из первых опытов с глобальным преобразованием в главе 7 было ясно, что к тексту можно применить те же эффекты масштабирования, вращения и сдвига, как и к любому другому графическому объекту. Программа RotatedFont, производная от класса *FontMenuForm*, рисует последовательность одинаковых текстовых строк по кругу вокруг центра экрана.

RotatedFont.cs

```
//---------------------------------------
// RotatedFont.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RotatedFont: FontMenuForm
{
     const int iDegrees = 20;       // 360 должно делиться на это число.

     public new static void Main()
     {
          Application.Run(new RotatedFont());
     }
     public  RotatedFont()
     {
          Text = "Rotated Font";

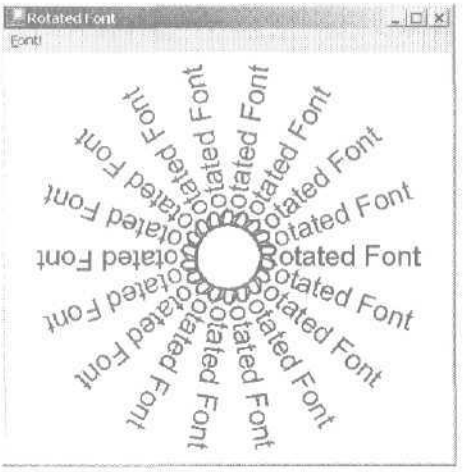
          strText = "   Rotated Font";
          font = new Font("Arial", 18);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush brush = new SolidBrush(clr);
          StringFormat strfmt = new StringFormat();
          strfmt.LineAlignment = StringAlignment.Center;

          grfx.TranslateTransform(cx / 2, cy / 2);

          for (int i = 0; i < 360; i += iDegrees)
          {
               grfx.DrawString(strText, font, brush, 0, 0, strfmt);
               grfx.RotateTransform(iDegrees);
          }
     }
}
```

см. след. стр.

Метод *DoPage* вызывает *TranslateTransform* чтобы выставить начальную точку в центре экрана. Затем он рисует 18 строк текста, каждая из которых повернута на 20° относительно предыдущей вокруг начальной точки. Метод *DrawString* использует объект *StnngFonnat,* который центрирует текстовую строку по вертикали. Текстовая строка начинается с трех пустых знаков, чтобы не было мешанины в центре. Вот результат работы программы с 18-пунктным шрифтом Arial:



Пункт меню «Font!» позволяет изменить шрифт на любой другой.

Вот программа, использующая метод *GetAscent* программы FontMenuForm. В главе 7 я показал, что отрицательные коэффициенты масштабирования приводят к зеркальному отражению графических объектов по горизонтали, вертикали или по обоим направлениям сразу. Программа ReflectedText выводит текстовую строку «Reflect», используя четыре возможных комбинации положительного и отрицательного масштабирования.

ReflectedText.cs

```
//  ___  ________________  __----------
// ReflectedText.cs © 2001 by Charles Petzold
//_____-____-------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ReflectedText: FontMenuForm
{
     public new static void Main()
     {
          Application.fiunfnewReflectedText());
     }
     public ReflectedTextO
     {
```

см. след. стр.

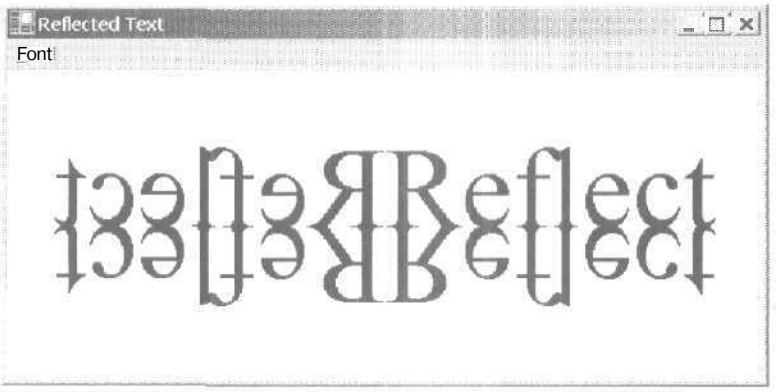
```
            Text = "Reflected Text";
            Width = 2;
            strText = "Reflect";
            font = new Font("Times New Roman", 54);
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            Brush brush = new SolidBrush(clr);
            float fAscent = GetAscent(grfx, font);
            StringFormat strfmt = StringFormat.GenericTypographic;

            grfx.TranslateTransform(cx / 2, cy / 2);

            for (int i = 0; i < 4; i++)
            {
                GraphicsState grfxstate = grfx.Save();

                grfx.ScaleTransform((i > 1 ? -1 : 1), (i & 1) == 1 ? -1 : 1);
                grfx.DrawString(strText, font, brush, 0, -fAscent, strfmt);
                grfx.Restore(grfxstate);
            }
        }
    }
```

Метод *TranslateTransform* устанавливает начальную точку в центре клиентской области. Метод *ScaleTransform* выглядит немного запутанным, но он всего лишь перебирает 4 комбинации из 1 и –1 для коэффициентов масштабирования при помощи переменной *i*. Аргумент *-fAscent* метода *DrawString* позиционирует текст, так чтобы левый край базовой линии совпадал с начальной точкой:



Вы можете также комбинировать эффекты. Вот та же программа с вызовом метода *RotateTransform* между *TranslateTransform* и *ScaleTransform*.

RotateAndReflect.cs

```
//-----------------------------------------
// RotateAndReflect.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class RotateAndReflect: FontMenuForm
{
     public new static void Main()
     {
          Application.Run(new RotateAndReflect());
     }
     public flotateAndfieflectQ
     {
          Text = "Rotated and Reflected Text";

          strText = "Reflect";
          font = new Font("Times New Roman", 36);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush brush   * new SolidBrush(clr);
          float fAscent = GetAscent(grfx, font);
          StringFormat strfmt = StringFormat.GenericTypographic;
          grfx.TranslateTransform(cx / 2, cy / 2);

          for (int i = 0; i < 4; i++)
          {
               GraphicsState grfxstate - grfx.Save();

               grfx.RotateTransform(-45);
               grfx.ScaleTransform((i > 1 ? -1 : 1), (i & 1} -- 1 ? -1 : 1);
               grfx.DrawString(strText, font, brush, 0, -fAscent, strfmt);
               grfx.Restore(grfxstate);
          }
     }
}
```

Я выбрал шрифт поменьше, чтобы изображение на экране не было слишком большим.

Поворот не искажает отдельные знаки. Поворачивая эту книгу из стороны в сторону, вы увидите, что строки выглядит совершенно нормально за исключением того, что они ориентированы в разных направлениях. Если же вы деформируете текстовую строку сдвигом, отдельные знаки *будут* искажены. Горизонтальные и вертикальные штрихи больше не будут находиться под правильными углами друг к другу.



Вот программа, которая устанавливает горизонтальный сдвиг равным 0,5, используя метод *Shear* класса *Matrix*.

SimpleShear.cs

```
//---------------------------------------
// SimpleShear.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class SimpleShear: FontMenuForm
{
     public new static void Main()
     {
          Application.Run(new SimpleShear());
     }
     public SimpleShear()
     {
          Text = "Simple Shear";

          strText = "Shear";
          font = new Font("Times New Roman", 72);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush brush = new SolidBrush(clr);
          Matrix matx = new Matrix();
          matx.Shear(0.5f, 0);
          grfx.Transform = matx;
          grfx.DrawString(strText, font, brush, 0, 0);
     }
}
```

Вызов метода *Shear* изменяет матрицу преобразования по умолчанию с:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

на:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

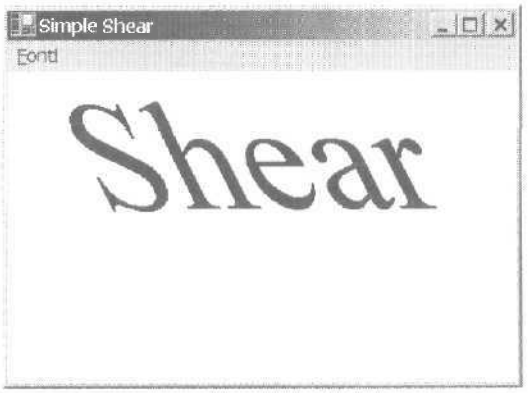
и формулы преобразования с:

$x' = x$
$y' = y$

на:

$x' = x + 0{,}5 \cdot y$
$y = y$

На самом верху клиентской области (или печатной страницы) y равно 0 и сдвига нет. Но при перемещении вниз по клиентской области, у возрастает и, следовательно, x' все больше сдвигается вправо. В результате мы получим строку:



Хотя знаки наклонены влево, они все еще находятся на горизонтальной базовой линии шрифта. Это значит, что знаки не просто повернуты, но и деформированы. Этот эффект является обратным наклонному начертанию (или курсиву). При увеличении первого аргумента *Shear* эффект становится более выраженным. Если сделать аргумент отрицательным, вы получите более привычный наклонный текст.

Если в методе *Shear* поменять местами аргументы:

```
matx.Shear(0, 0.5f);
```

вы получите вертикальный сдвиг (см. рис. на следующей странице).

Видно, что вертикальные штрихи в буквах *h* и *r* все еще вертикальные, но базовая линия теперь находятся под углом к горизонтальной оси,



Сдвиг текста можно задействовать более хитроумно. Допустим, вы хотите нарисовать текст на базовой линии шрифта, как в программе TextOnBaseline в главе 9:



Далее предположим, вы хотите придать тексту более изощренный вид, используя сдвиг. Возможно, вы захотите, чтобы текст оставался на той же самой базовой линии шрифта в том же месте, но был слегка наклонен вперед. Поскольку сдвиг производится всегда относительно начальной точки, нужно переместить ее на базовую линию шрифта,

Программа BaselineTilt очень похожа на программу TextOnBaseline, но является производной от FontMenuForm. BaselineTilt отличается и тем, что устанавливает базовую линию шрифта на расстоянии 3/4 от верха клиентской области, а не посредине (чтобы лучшим образом задействовать клиентскую область) и использует метод *GetAscent* в FontMenuForm,

BaselineTilt.cs

```
//-------------------------------------------
// BaselineTilt.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
```

см. след. стр.

```
using System.Windows.Forms;

class BaselineTilt: FontMenuForm
{
     public new static void Main()
     {
          Application.Run(new BaselineTilt());
     }
     public BaselineTilt()
     {
          Text = "Baseline Tilt";

          strText = "Baseline";
          font = new Font("Times New Roman", 144);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          float yBaseline = 3 * cy / 4;
          float cyAscent = GetAscent(grfx, font);

          grfx.DrawLine(new Pen(clr), 0, yBaseline, cx, yBaseline);

          grfx.TranslateTransform(0, yBaseline);

          Matrix matx = grfx.Transform;
          matx.Shear(-0.5f, 0);
          grfx.Transform = matx;

          grfx.DrawString(strText, font, new SolidBrush(clr), 0, -cyAscent);
     }
}
```

Эта программа также деформирует текстовую строку сдвигом. Метол *Translate-Transform* устанавливает начальную точку па нужной базовой линии на левой с гороне клиентской области:

```
grfx.TranslateTransform(0, yBaseline);
```

Следующие три метода устанавливают отрицательный горизонтальный сдвиг

```
Matrix matx = grfx.Transform;
matx.Shear(-0.5f, 0);
grfx.Transform = matx;
```

Предшествующий вызов метода *TranslateTransform* приводит к тому, что сдвиг производится относительно новой начальной точки. Вот формула трансформирования:

$$x' = x - 0{,}5 \cdot y$$
$$y' = y + yBaseline$$

Точка (0, 0) в глобальных координатах соответствует точке (0, *yBaseline*) в координатах клиентской области, которая указывает на искомую базовую линию шр: ф-

та. Пункты выше базовой линии (глобальные координаты с отрицательной величиной *y)* сдвигаются вправо. Точки ниже базовой линии (положительная величина *y)* сдвигаются влево.

Программа выводит текст в точке с глобальными координатами (0. *–cyAscent*), где *cyAscent* — это высота знаков над базовой линией шрифтов. Я использую метод *DrawString* без аргумента *StringFormat*, чтобы заданные глобальные координаты указывали на точку, соответствующую верхнему левому углу текстовой строки. Формула преобразования ставит в соответствие координате (0. *–cyClient*) точку (0.5 ' *cyAscent*, *y Baseline - cyAscent)*. Левый конец базовой линии шрифта текстовой строки выводится в точке (0, 0) в глобальных координатах, которая (как я говорил) соответствует точке (0, *yBaseline*), что нам и нужно:



Хотя это кажется курсивом, на самом деле это не так: строчная буква *a* имеет латинское начертание, а не курсивное.

Теперь, зная, как вывести нормальный текст и текст со сдвигом на одной базовой линии шрифта, мы получаем в свое распоряжение очень интересную методику работы со шрифтом. Вот программа, которая демонстрирует эту методику, увеличивающую сдвиг и растягивающую сдвинутый текст по высоте:

TiltedShadow.cs

```
//-----------------------------------------
// TiltedShadow.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TiltedShadow: FontMenuForm
 {
     public new static void Main()
     {
          Application.Run(new TiltedShadow());
     }
     public TiltedShadowO
     C
          Text * "Tilted Shadow";

          strText * "Shadow";
```

см. след. стр.

```
            font = new Font("Times New Roman", 54);
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            float fAscent = GetAscent(grfx, font);

                // Устанавливаем базовую линию на расстоянии 3/4 от верха
                // клиентской области.

            grfx.TranslateTransform(0, 3 * cy / 4);

                // Сохраняем состояние графики.

            GraphicsState grfxstate = grfx.Save();

                // Устанавливаем масштабирование, сдвиг и падающую тень.

            grfx.MultiplyTransform(new Matrix(1, 8, -3, 3, 0, 0»;
            grfx.DrawString(strText, font, Brushes.DarkGray, 0, -fAscent);

                // Рисуем текст без масштабирования и сдвига.

            grfx.Restore(grfxstate);
            grfx.DrawString(strText, font, Brushes.Black, 0, -fAscent);
        }
    }
```

Как и BaselineTilt, программа TiltedShadow использует метод *TranslateTransform*, чтобы установить начальную точку с координатами клиентской области *(0, Усу/4)*. Метод *MultiplyTransform* затем умножает результаты на матрицу:

$$\begin{vmatrix} 1 & 0 & 0 \\ -3 & 3 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

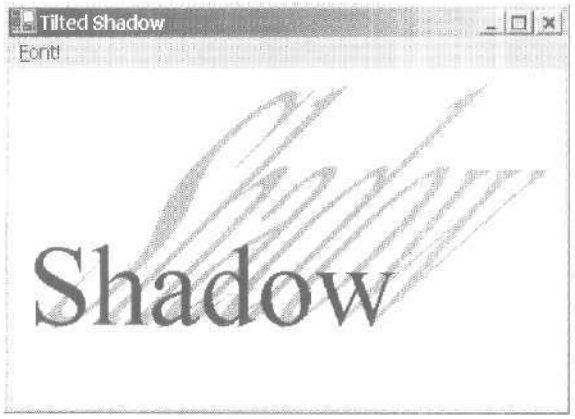
Результирующее составное преобразование определяется формулами:

$x' = x - 3 \cdot y$
$y = 3 - y + (3 * cy / 4)$

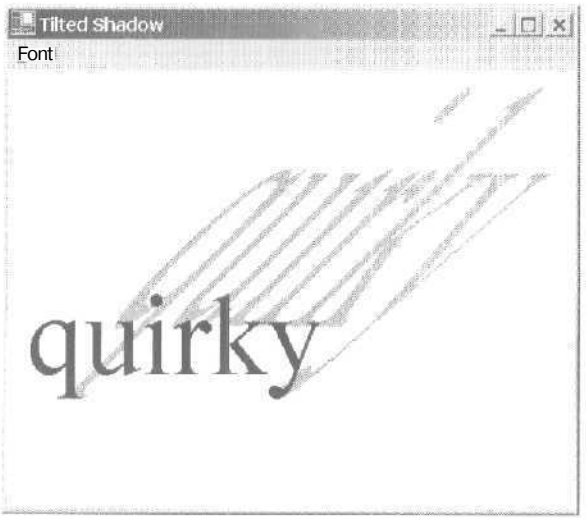
Теперь сдвиг больше, чем в предыдущей программе, а координата *y* утраивается. В сочетании с нормальным текстом на базовой линии шрифта мы получим эффект падающей тени (см. рис. на следующей странице).

Этот эффект имеет один недостаток. Когда в тексте есть знаки с нижними выносными элементами, тень ложится перед текстом. Чтобы тень начиналась от нижней части знаков, присвойте переменной, которую я назвал *fAscent*, сумму величин высоты с надстрочным элементом и подстрочного элемента:

```
float fAscent = GetAscent(grfx, font) + GetDescent(grfx, font);
```



Конечно, вы захотите изменить переменную *strlext,* чтобы задействовать соответствующую текстовую строку;



Заметьте: тень начинается от нижней части букв *q* и *y*.

Текст и контуры

Класс *GraphicsPath* включает метод *AddString*, позволяющий добавлять текстовую строку в контур. Прямые и кривые, из которых состоят очертания знаков, становятся частью контура. Как обычно, текст немного отличается от других графических объектов, и добавление текста в контур требует специального рассмотрения,

Первая проблема — в самом методе *AddString*. Как вы знаете из главы 15, большинство методов *Add* в *GraphicsPath* похожи на соответствующие им методы *Draw* в классе *Graphics*. Так, используя класс *Graphics*, можно нарисовать линию:

```
grfx.DrawLine(pen, x1, y1, x2, y2);
```

или добавить линию в контур, вызвав метод:

```
path.AddLine(x1, y1, x2, y2);
```

Метод *AddLine* не требует аргумента *Pen,* так как контур содержит только координаты линий.

Метод *AddString* класса *GraphicsPath* существенно отличается от метода *DrawString* класса *Graphics.* Вместо шрифта (как в *DrawString*) вы задаете три основных компонента, используемые при создании шрифта (гарнитура, начертание и размер), а также расположение (либо точка, либо прямоугольник) и объект *StringFormat*:

Методы *AddString* класса *GraphicsPath*

```
AddString(string str, FontFamily ff, int iStyle, float fSize, Point pt,
          StringFormat sf)
AddString(string str, FontFamily ff, int iStyle, float fSize, PointF ptf,
          StringFormat sf)
AddString(string str, FontFamily ff, int iStyle, float fSize,
          Rectangle rect, StringFormat sf)
AddString(string str, FontFamily ff, int iStyle, float fSize,
          RectangleF rectf, StringFormat sf)
```

Заметьте: третий аргумент определяется как *int,* но на самом деле — это член перечисления *FontStyle,* приведенный к типу *int.*

Создавая шрифты для рисования, вы обычно основываетесь на определенном метрическом размере. Очень часто размер шрифта определяется в пунктах, но можно применять дюймы и миллиметры. Как я показал в главе 9, можно создать шрифт, который задается *не* в конкретном метрическом размере, а использует значение *GraphicsUnit.Pixel* или *GraphicsUnit.World* в конструкторе *Font.* Размер такого шрифта — просто число. Когда вы преобразуете текст с таким шрифт >м, размеры приводятся к координатам текущей страницы выходного устройства.

Контур не содержит метрической информации — это лишь набор координат. Поэтому *AddString* нельзя определить в терминах *Font* (И если бы существовал метод *AddString,* который *действительно* включал бы аргумент *Font,* он, конечно, был бы ограничен объектом *Font,* созданным с помощью *GrapbicsUnitPixel* или *GraphicsUnit.World*).

Допустим, вы добавляете текстовую строку в контур, вызвав *AddString* с четвертым аргументом, равным 72. Что это число означает? Какой будет размер шрифта? Все зависит от преобразования страницы на момент визуализации контура методами *DrawPath* или *FillPath.* Если единицы измерения страницы определены как *GraphicsUnit.Point,* текст будет визуализирован с теми же размерами, как 72-пунктный шрифт, нарисованный с помощью *DrawString.* Но если вы выводите контур на принтер, используя единицы по умолчанию *GraphicsUnit.Display,12* единицы будут интерпретироваться как 0,72 дюйма, и текст будет преобразован как 52-пунктный шрифт. (То есть 0,72 дюйма умножить на 72 пункта в дюйме.) Если вы выводите контур на экран, используя единицы страницы по умолчанию *GraphicsUnit.Pixel,* размеры шрифта будут зависеть от разрешения монитора. Если разрешение монитора равно 120 dpi, то, например, шрифт в 72 единицы будет интерпретироваться как 72/120 дюйма и будет иметь размер примерно 43 пункта. (То есть 72/120 дюйма умножить на 72 пункта в дюйме.)

Вполне возможно, вы захотите, чтобы текст в контуре был совместим с текстом, выводимым обычным образом. Наверное, проще всего — использовать метод *AddString* с требуемым кеглем шрифта. Перед преобразованием контура вы просто устанавливаете единицы страницы в пункты:

```
grfx.PageUnit = GraphicsUnit.Point;
```

Другой подход (я использую его далее в этой главе) позволяет задействовать единицы страницы по умолчанию при визуализации шрифта. Но при этом вам нужно вычислить аргумент *fSize* для метода *AddString* на основе требуемого кегля шрифта и разрешения устройства вывода. (Это значит, что вы не можете использовать один и тот же контур, как для экрана, так и для принтера.) Пусть у вас уже есть объект *font* класса *Font,* и вы хотите добавить текст в контур на основе этого шрифта. Второй аргумент *AddString* — просто *font.FontFamily*. Третий аргумент — *font.FontStyle*, приведенный к типу *int*. Четвертый аргумент вычисляется так:

```
if (grfx.PageUnit == GraphicsUnit.Display)
     fFontSize = 100 * font.SizeInPoints / 72;
else
     fFontSize = grfx.DpiX * font.SizeInPoints / 72;
```

Первое вычисление производится для принтера, второе — для экрана. Если вернуться к программе FontMenuForm, вы увидите, что я уже использовал этс вычисление в методе *PointsToPageUnits*.

Независимо от того, как тщательно вы будете рассчитывать размеры шрифта, текст, выводимый методом *DrawString*, и текст, выводимый при визуализации контура, для которого использовался вызов *AddString,* не будут выглядеть абсолютно одинаково, особенно на экране. При вызове метода *DrawString* текст несколько изменяется — становится читабельней. Если нужно отобразить два фрагмента текста одинакового размера (например, если один накладывается на другой) не используйте контур для одного и метод *DrawString* для другого. Примените контур для обоих фрагментов.

Я уже упоминал о *разметке* (hints) при рассмотрении контурных шрифтов. Когда вы добавляете текстовую строку в контур, все, что сохраняется, — это координаты с плавающей запятой. Разметки больше нет. Когда контур визуализируется, координаты с плавающей запятой должны быть переведены в пикселы. Будет применено некоторое округление, и оно не будет вполне согласованным. Так, ширина двух вертикальных штрихов в знаке *H* может отличаться на пиксел. Для шрифтов с большими размерами вы не заметите разницы. На устройствах с большим разрешением, таких как принтер, тоже не будет проблемы. Но для нормальных размеров шрифтов на экране текст будет иметь неприемлемый вид,

По этой причине текст, добавляемый в контур, выглядит лучше с большими размерами шрифтов или на устройствах с высоким разрешением, таких как принтеры. *Применяйте контур только для специальных текстовых эффектов, избегайте использования контура для обычного текста.*

Рассмотрим пример. Некоторые графические системы позволяют создавать *оконтуренные* (outline) или *пустотелые* (hollow) шрифты, состоящие из одних контуров, не заполненных внутри. Windows Forms не включает такой стиль в перечисление *FontStyle*. Чтобы вывести знаки пустотелых шрифтов, можете использовать контур, как я сделал это в программе HollowFont.

HollowFont.cs

```
//---------------------------------------
// HollowFont.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HollowFont: FontMenuForm
{
     public new static void Main()
     {
          Application.Run(new HollowFont());
     }
     public HollowFont()
     {
          Text = "Hollow Font";
          Width *= 2;
          strText = "Hollow";
          font - new Font("Times Mew floman". 108);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          GraphicsPath path - new GraphicsPath();
          float fFontSize = PointsToPageUnits(grfx, font);

               // Получаем координаты для размещения строки по центру.

          SizeF sizef = grfx.MeasureString(strText, font);
          PointF ptf « new PointF((cx - sizef.Width) / 2,
                                  (cy - sizef.Height) / 2);

               // Добавляем текст а контур.

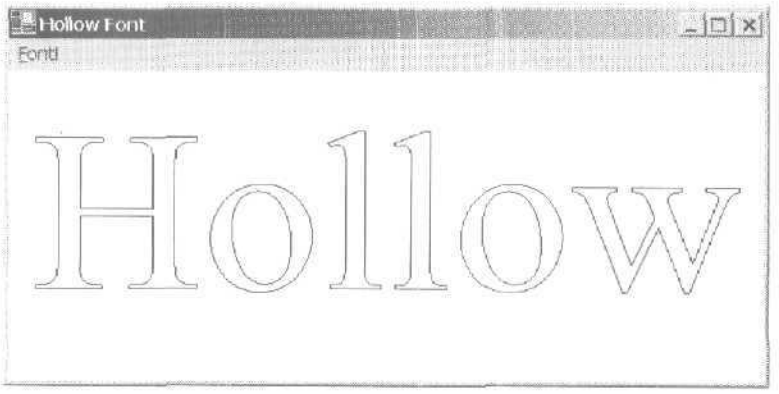
          path.AddString(strText, font.FontFamily, (int) font.Style,
                         fFontSize, ptf, new StringFormat());

               // Рисуем контур.

          grfx.DrawPath(new Pen(clr), path);
     }
}
```

Перед вызовом метода *AddString* программа вычисляет размер шрифта (он хранится в переменной *fFontSize*), используя описанную выше методику. Программа также вычисляет аргумент *PointF* метода *AddString*, который располагает строку в центре клиентской области при выводе контура (что происходит в конце метода *DoPage*).

Программа HollowFont вычисляет место для размещения текста *PointF* перед добавлением текста в контур, потому что метод *DrawPath* не имеет аргумента, показывающего, где нарисован контур. При вызове *DrawPath* все координаты в контуре интерпретируются как глобальные. Аргумент *PointF* метода *AddString* указывает на координаты верхнего левого угла текстовой строки. Все координаты знаков текста вычисляются относительно этой точки, и именно эти координаты хранятся в контуре. Программа HollowFont вычисляет эту точку; используя *MeasureString* с начальным объектом *Font,* как если бы она собиралась выводить текст с помощью *DrawString.* Вот результат:



Щелкните меню «Font!» и выберите 12-пунктный шрифт. Вы увидите, что на мониторе текст читается плохо. Это проблема округления.

Есть и другой подход к центрированию текста, хранимого в контуре. Этот альтернативный подход более общий, так как основывается на координатах, хранимых в самом контуре. Метод *GetBounds* класса *GraphicsPath* возвращает *RectangleF,* который представляет собой минимальный прямоугольник, заключающий в себе все координаты контура. Вы можете использовать этот прямоугольник, чтобы центрировать содержимое контура. Вероятно, проще сделать это, вызвав метод *TranslateTransform* объекта *Graphics.*

Не думайте, что верхний левый угол прямоугольника, возвращаемого методом *GetBounds,* будет располагаться в точке (0, 0). даже если вы вызовете метод *AddString* с аргументом *PointF,* равным (0, 0). Свойство *Left* прямоугольника будет, вероятно, немного больше 0, так как обычно оставляется небольшое поле перед первым знаком текста. Свойство *Top* прямоугольника тоже будет немного больше 0.

Эта программа вызывает метод *AddString* с аргументом *PointF,* равным (0, 0), и затем центрирует контур в клиентской области, используя ограничивающий его прямоугольник:

HollowFontCenteredPath.cs

```
//-----------------------------------------------------
// HollowFontCenteredPath.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using System;
```

см. след. стр.

```
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HollowFontCenteredPath: FontMenuForm
{
     public new static void Main()
     {
          Application.Run(new HollowFontCenteredPath());
     }
     public HollowFontCenteredPath()
     {
          Text = "Hollow Font (Centered Path)";
          Width *= 2;
          strText = "Hollow";
          font - new Font("Times New Roman", 108);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     Ч
          GraphicsPath path = new GraphicsPath();
          float fFontSize * PointsToPageUnits(grfx, font);

               // Добавляем текст в контур.

          path.AddString(strText, font.FontFamily, (int) font.Style,
                         fFontSize, new PointF(0, 0), new StringFormat());

               // Получаем границы контура для центрирования.

          RectangleF rectfBounds = path.GetBounds();

          grfx.TranslateTransform(
                         (cx - rectfBounds.Width) / 2 - rectfBounds.Left,
                         (cy - rectfBounds.Height) / 2 - rectfBounds.Top);

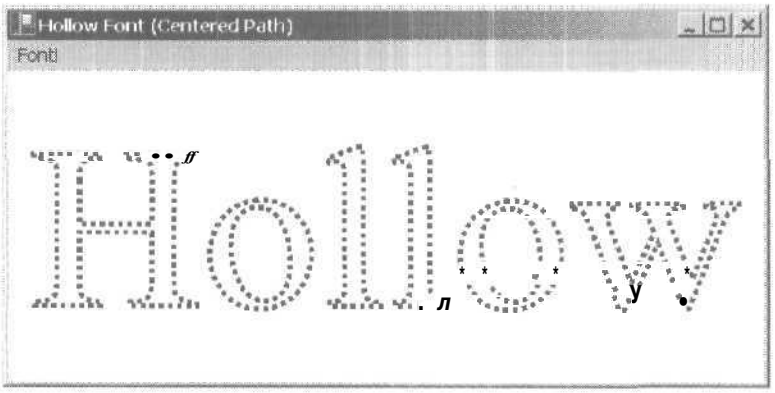
            ;, // Рисуем контур.

          Pen pen = new Pen(clr, fFontSize / 50);
          pen.DashStyle - DashStyle.Dot;

          grfx.DrawPath(pen, path);
     }
}
```

Метод *TranslateTransform* использует ширину и высоту области экрана, а также координаты верхнего левого угла и размеры ограничивающего прямоугольника.

Здесь я несколько иначе определил перо, на этот раз сделав толщину линии равной 1/50 от *fFontSize* и установив *DashStyle* в *Dot*.



Если сравнить изображение экрана с картинкой из программы HollowFont, вы увидите, что оно центрировано иначе. Центрирование по вертикали текста в HollowFont основано на высоте, возвращаемой методом *MeasureString*. Эта высота связана со шрифтом и включает нижние выносные элементы и диакритические знаки, даже если их нет в конкретной измеряемой текстовой строке. Так, *MeasureString* возвращает одинаковую высоту для строк «A», «a» и «y».

Текст, выводимый HollowFontCenteredPath, центрируется по вертикали исключительно на основе координат в контуре. Центрирование выглядит аккуратнее.

Вы помните программу HowdyWorldFullFit из главы 9, которая пыталась увеличить строку, используя *MeasureString* и *ScaleTransform*,чтобы подогнать ее размеры под прямоугольник? Это было несложно, потому что *MeasureString* включает поля для нижних выносных элементов и диакритических знаков. Вот программа, в которой ограничивающий контур прямоугольник и *ScaleTransform* служат для плотного заполнения клиентской области короткой текстовой строкой.

FullFit.cs

```
//---------------------------------------
// FullFit.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
class FullFit: FontMenuForm
{
     public new static void Main()
     {
          Application.Run(new FullFit());
     }
     public FullFit()
     {
          Text = "Full Fit";

          strText = "Full Fit";
```

см. след. стр.

```
            font = new Font("Times New flostan", 108);
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            GraphicsPath path = new GraphicsPath();

                // Добавляем текст в контур.

            path.AddString(strText, font.FontFamily, (int) font.Style,
                           100, new Point(0, 0), new StringFormat());

                // Устанавливаем глобальное преобразование.

            RectangleF rectfBounds = path.GetBounds();
            PointF[] aptfOest = { new PointF(0, 0), new PointF(cx, 0),
                                                    new PointFCO, cy) };

            grfx.Transform = new Matrix(rectfBounds, aptfDest);

                // Заполняем контур.

            grfx.FillPath(new SolidBrush(clr), path);
        }
    }
```

Программа устанавливает глобальное преобразование, используя мощный конструктор *Matrix,* у которого два аргумента: структура *RectangleF* и массив из трех структур *PointF*:

```
grfx.Transform = new Matrix(rectfBounds, aptfDest);
```

Этот конструктор вычисляет преобразование, которое сопоставляет трем углам структуры *RectangleF* три структуры *PointF*. Просто установите структуру *RectangleF* равной прямоугольнику, ограничивающему контур, и три структуры *PointF* равными трем углам области дисплея, и контур заполнит все пространство.



Вы, конечно, захотите распечатать текст. Вы можете изменить шрифт, но размер задаваемого шрифта не имеет значения: текст всегда масштабируется под ра ме-

ры клиентской области. Поэтому метод *AddString* вызывается с произвольно выбранным размером шрифта 100.

Я уже говорил, что, когда вы хотите вывести текст с помощью контура, вы также должны использовать его для любого другого текста, который должен иметь соответствующий размер, даже если для другого текста контур не нужен. Следующая программа применяет один контур для двух вызовов метода *FillPath* и для одного вызова метода *DrawPath*. Два вызова *FillPath* могли бы быть сделаны посредством вызова *DrawString*, но при этом текст не был бы правильно выровнен на экране.

DropShadowWithPath.cs

```
//-----------------------------------------------
// DropShadowWithPath.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class DropShadowWithPath: FontMenuForm
{
     const int iOffset = 10;  // Примерно (для принтера точно) 1/10 дюйма

     public new static void Main()
     {
          Application.Run(new DropShadowWithPath());
     }
     public DropShadowWithPath()
     {
          Text = "Drop Shadow with Path";
          Width *= 2;
          strText = "Shadow";
          font = new Font("Times New Roman", 106);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          GraphicsPath path = new GraphicsPath();
          float fFontSize = PointsToPageUnits(grfx, font);

               // Получаем координаты для размещений строки по центру.

          SizeF sizef = grfx.MeasureString(strText, font);
          PointF ptf * new PointF((cx - sizef.Width) / 2,
                                  (cy - sizef.Height) / 2);

               // Добавляем текст в контур.

          path.AddString(strText, font.FontFamily, (int) font.Style,
```

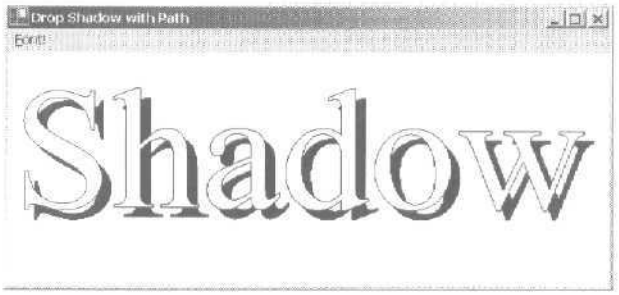
см. след. стр.

```
                    fFontSize, ptf, new StringFormat());

                // Очищаем, заполняем, переводим, заполняем и рисуем.

            grfx.Clear(Color.White);
            grfx.FillPath(Brushes.Black, path);
            path.Transform(new Matrix(1, 0, 0, 1, -10, -10»;
            grfx.FillPath(Brushes.White, path);
            grfx.DrawPath(Pens.Black, path);
        }
}
```

Знаки шрифта при оконтуривании можно закрасить в фоновый цвет. Мне больше нравится такой вид падающей тени, чем предыдущий;



Аналогичную методику можно использовать для объемного эффекта на белом ф же.

Вот программа, которая рисует контуры знаков шрифта, используя перо, созданное из штриховой кисти с толщиной, составляющей 1/20 от величины *fFontSize*:

HollowFontWidePen.cs

```
//-----------------------------------------------
// HollowFontWidePen.cs © 2001 by Charles Petzolcf
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HollowFontWidePen: FontMenuForm
{
    public new static void Main()
    {
        Application. flunks» HollowFontWidePen());
    }
    public HollowFontWidePen()
    {
        Text = "Hollow Font (Wide Pen)";
        Width *= 2;
        strText = "Wide Pen";
```

см. след. стр.

```
            font = new Font("Times New Roman", 108,
                            FontStyle.Bold | FontStyle.Italic);
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            GraphicsPath path = new GraphicsPath();
            float fFontSize = PointsToPageUnits(grfx, font);

                // Добавляем текст в контур.

            path.AddString(strText, font.FontFamily, (int) font.Style,
                           fFontSize, new PointF(0, 0), new StringFormat());

                // Получаем границы контура для центрирования.

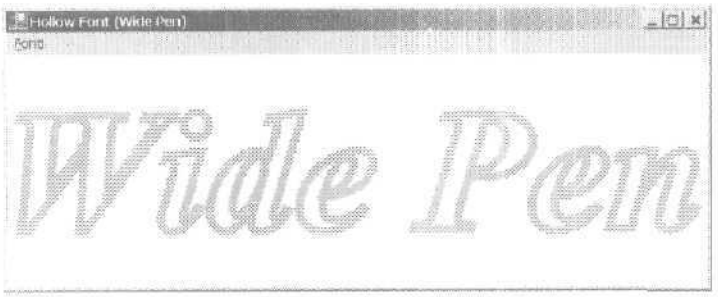
            RectangleF rectfBounds = path.GetBounds();

            grfx.TranslateTransform(
                           (cx - rectfBounds.Width) / 2 - rectfBounds.Left,
                           (cy - rectfBounds.Height) / 2 - rectfBounds.Top);

                // Рисуем контур.

            Brush brush = new HatchBrush(HatchStyle.Trellis,
                                         Color.White, Color.Black);
            Pen pen = new Pen(brush, fFontSize / 20);
            grfx.DrawPath(pen, path);
        }
    }
```

Этот эффект меня не вполне удовлетворяет:



По-моему, не хватает границы вокруг нарисованных кистью знаков. Иначе говоря, контуры шрифтов сами должны быть обведены. Возможно ли это? Да, именно это происходит, когда вы вызываете метод *Widen* класса *GraphicsPath*. Новый контур создается на основе существующего, как если бы это было нарисовано пером с определенной толщиной линии. Взгляните на такую программу:

HollowFontWidened.cs

```
//-------------------------------------------
// HollowFontWidened.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HollowFontWidened: FontMenuForm
{
     public new static void Main()
     {
          Application.Run(new HollowFontWidened());
     }
     public  HollowFontWidened()
     {
          Text = "Hollow Font (Widened)";
          Width *= 2;
          strText * "Widened";
          font = new Font("Times New Roman", 108,
                          FontStyle.Bold | FontStyle.Italic);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          GraphicsPath path * new GraphicsPath();
          float fFontSize = PointsToPageUnits(grfx, font);

               // Добавляем текст в контур.
          path.AddString(strText, font.FontFamily, (int) font.Style,
                         fFontSize, new PointF(0, 0), new StringFormat());

               // Получаем границы контура для центрирования.

          flectangleF rectfBounds * path.GetBounds();

          grfx.TranslateTransform(
                         (ex - rectfBounds.Width) / 2 - rectfBounds.Left,
                         (cy - rectfBounds.Height) / 2 - rectfBounds.Top);

               // Обводим, заполняем и рисуем контур.

          path.Widen(new Pen(Color.Black, fFontSize / 20));
          Brush brush * new HatchBrush(HatchStyle.Trellis,
                                       Color.White, Color.Black);
          grfx.DrawPath(new Pen(Color.Black, 2), path);
          grfx.FillPath(brush, path);
     }
}
```

Весь новый код добавляется в конце тела метода *DoPage*. Предыдущая программа вызывала метод *DrawPath*, используя толстое перо, основанное на штриховой кисти. Новая программа рисует в основном то же самое, используя метод *Widen*, основанный на толстом пере, а затем метод *FillPath* со штриховой кистью.

Вы можете убедиться в этом, закомментировав вызов метода *DrawPath* в HollowFontWidened. Но *DrawPath* является методом, который обеспечивает эффект к которому я стремился. Он очерчивает контуры:



Я пришел к некоторым программным решениям в HollowBrushWidened путем экспериментов. Сначала я вызывал метод *DraivPatb* после *FillPatb*, но алгоритм расширения контура приводил к тому, что *DrawPatb* рисовал какие-то закорючки. (Попробуйте убрать метод *FillPatb*, чтобы увидеть, о чем я говорю. Вам даже может понравиться этот эффект, похожий на результат рисования слегка расстроенной механической рисовальной машины.) Я переместил *FillPath* за *DrawPath*, чтобы спрятать закорючки, и был вынужден применить перо шириной 2 в методе *DrawPatb*, потому что кисть затушевывала линию пера.

Вы можете также установить область, вырезанную из контура, т. е. вы можете ограничить графический вывод внутренней частью текстовой строки. Бот программа, которая устанавливает вырезанную область и затем рисует кривые Безье цветами, выбранными случайным образом:

ClipText.cs

```
//-------------------------------------------
// ClipText.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
tiding System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ClipText: FontMenuForm
<
    public new static void Main()
    {
        Application.Run(new ClipText());
    }
    public ClipTextO
```

см. след. стр.

```
    {
        Text = "Clip Text";
        Width *= 2;
        strText = "Clip Text";
        font = new Font("Times New Roman", 108, FontStyle.Bold);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        GraphicsPath path = new GraphicsPath();
        float fFontSize = PointsToPageUnits(grfx, font);

            // Добавляем текст в контур,

        path.AddString(strText, font.FontFamily, (int) font.Style,
                       fFontSize, new PointF(0, 0), new StringFormat());

            // Устанавливаем вырезанную область.

        grfx.SetClip(path);

            // Получаем границы контура и центр вырезанной области,

        RectangleF rectfBounds = path.GetBounds();
        grfx.TranslatedIp(
                       <cx - rectfBounds.Width) / 2 - rectf Bounds. Left,
                       <cy - rectfBounds.Height) / 2 - rectfBounds.Top);

            // Рисуем обрезанные линии.

        Random rand - new Random(); ;

        for (int y = 0; y < cy; y++)
        {
            Pen pen = new Pen(Color.FromArgb(rand.Next(255),
                                             rand.Next(255),
                                             rand.Next(255)));

            grfx.DrawBezier(pen, new Point(0, y),
                                 new Point(cx / 3, y + cy / 3),
                                 new Point(2 * cx / 3, y - cy / 3),
                                 new Point(cx, y));
        }
    }
}
```

Вот результат:



Нелинейные преобразования

Матричное преобразование широко используется в графической системе Windows Forms. Вы можете применить его к объекту *Graphics*, к контуру, к кистям и перьям. Но матричное преобразование — это всегда *линейное* преобразование. Параллельные линии всегда преобразуются в другие параллельные линии, и через какое-то время вы обнаруживаете, что эти параллельные линии напоминают вам прутья на тюремных решетках.

Класс *GraphicsPath* имеет одно нелинейное преобразование, доступное через метод *Warp*. Следующая программа сохраняет некоторый текст в контуре и затем использует *Warp* для сжатия верхней части контура.

WarpText.cs

```
//-------------------------------------------
// WarpText.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class WarpText: FontMenuForm
{
     int iWarpMode * 0;

     public new static void Main()
     {
          Application.Run(new WarpText());
     }
     public WarpText()
     {
          Text = "Warp Text - " + (WarpMode) iWarpMode;
          Menu.MenuItems.Add("&Toggle!",
                             new EventHandler(MenuToggleOnClick));
          StrText = "WARP";
          font - new Font("Arial Black", 24);
     }
     void MenuToggleOnClick(object obj, EventArgs ea)
```

см. след. стр.

```
    {
        iWarpMode ^= 1;
        Text * "Warp Text - " + (WarpMode) iWarpMode;
        Invalidate();
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        GraphicsPath path * new GraphicsPath();

            // Добавляем текст в контур.

        path.AddString(strText, font.FontFamily, (int) font.Style,
                    100, new PointF(0, 0), new StringFormat());

            // Преобразовываем контур.

        flectariQleF rectfBounds * path.GetBounds();
        PointF[]   aptfDest    = { new PointF(cx / 3, 0),
                                   new PointF(2 * cx / 3, 0),
                                   Re» PointF(0, cy),
                                   new PointF(cx, cy) И

        path.Warp(aptfDest, rectfBounds, new Matrix(),
                  (WarpMode) iWarpMode);

            // Заполняем контур.

        grfx.FillPath(new SolidBrush(clr), path);
    }
}
```

В программе FullFit конструктор *Matrix* вычислял преобразование, сопоставляющее трем углам прямоугольника 3 точки в массиве. Метод *Warp* работает аналогично, но он сопоставляет 4 углам прямоугольника (я вновь использую ограничивающий контур прямоугольник) 4 точки массива. В качестве конечного массива я использовал два нижних угла клиентской области и 2 точки наверху клиентской области, расположенные ближе к центру, чем верхние углы.



Пункт меню «Toggle!» позволяет переключаться между перспективным и билинейным режимами преобразования.

Хотя метод *Warp* — единственное нелинейное преобразование, непосредственно доступное программистам в Windows Forms, графические контуры позволяют совершать любые нелинейные преобразования, которые вы можете придумать и описать математически. Вот как это сделать:

- получить массив координат в контуре, используя свойство *PathPoints*;
- изменить эти координаты, используя необходимые формулы преобразования;
- создать новый контур на основе измененных координат.

Конечно, самое трудное — это придумать формулы преобразования.

Следующая программа выводит строку текста, расширяющуюся в средней части.

TallInTheCenter.cs

```
//-----------------------------------------
// TallInTheCenter.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TallInTheCenter: FontMenuForm
{
     public new static void Main ()
     {
          Application.Run(new TallInTheCenter());
     }
     public TallInTheCenter()
     {
          Text = "Tall in the Center";
          Width *= 2;
          strText = Text;
          font = new Font("Times New Roman", 48);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          GraphicsPath path = new GraphicsPath();
          float fFontSize = PointsToPageUnits(grfx, font);

               // Добавляем текст в контур.

          path.AddString(strText, font.FontFamily, (int) font.Style,
                         fFontSize, new PointF(0, 0), new StringFormat());

               // Сдвигаем начальную точку в центр контура.

          RectangleF rectf = path.GetBounds();
```

см. след. стр.

```
            path.Transform(new Matrix(1, 0, 0, 1,
                                      -(rectf.Left + rectf.Right) / 2,
                                      -(rectf.Top + rectf.Bottom) / 2)5;
            rectf = path.GetBounds();

                // Изменяем контур,

            PointF[] aptf = path.PathPoints;

            for (int i = 0; 1 < aptf.Length; i++)
                aptf[i].Y *= 2 * (rectf.Width - Math.Abs(aptf[i].X)) /
                                                                rectf.Width;
            path * new GraphicsPath(aptf, path.PathTypes);

                // Заполняем контур.

            grfx.TranslateTransform(cx / 2, cy / 2);
            grfx.FillPath(new SolidBrush(clr), path);
        }
    }
```

Одна из полезных методик в подобных программах состоит в подготовке контура к нелинейному преобразованию путем предварительного линейного преобразования. Получив границы контура (хранимые в переменной *rectf*), программа сдвигает начальную точку к центру контура:

```
path.Transform(new Matrix(1, 0, 0, 1,
                          -(rectf.Left + rectf.Right) / 2,
                          -(rectf.Top + rectf.Bottom) / 2));
```

Затем она вновь вызывает метод *GetBounds*, чтобы сохранить новые границы в *rectf*.

Чтобы начать нелинейное преобразование, программа получает массив структур *PointF*, определяемых контуром:

```
PointF[] aptf = path.PathPoints;
```

Далее программа изменяет эти значения, увеличивая координату Y в зависимости от того, насколько близко точка находится к центру:

```
for (int i = 0; i < aptf.Length; i++)
    aptf[i].Y *= 2 * (rectf.Width - Math.Abs(aptf[i].X)) / rectf.Width;
```

Затем она создает новый контур:

```
path = new GraphicsPath(aptf, path.PathTypes);
```

Вот как это будет выглядеть:



Как и в программе FullFit, начальные размеры шрифта не имеют значения.

Вот другой пример: программа WrapText (не путайте с WarpText) поворачивает текстовую строку вдоль окружности. Начальный размер шрифта тоже не имеет значения, но программе нужно знать высоту знаков с надстрочным элементом, поэтому вызов метода *AddString* основан на конкретном шрифте.

WrapText.cs

```
//------------------------------------
// WrapText.cs © 2001 by Charles Petzold
//------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class WrapText: FontMenuForm
{
     float fRadius = 100;

     public new static void Main()
     {
          Application.Run(new WrapText());
     }
     publicWrapTextO
     {
          Text = "Wrap Text";

          strText = "e snake ate the tail of th";
          font * new Font("Times New Roman", 48);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          GraphicsPath path = new GraphicsPath();
          float fFontSize = PointsToPageUnits(grfx, font);
```

см. след. стр.

```
            // Добавляем текст в контур.

            path.AddString(strText, font.FontFamily, (int) font.Style,
                           fFontSize, new PointF(0, 0), new StringFormat());

            // Сдвигаем начальную точку к левому краю базовой линии,
            // увеличивая у,

            RectangleF rectf = path.GetBounds();
            path.Transform(new Matrix(1, 0, 0, -1, -rectf.Left,
                                      GetAscent(grfx, font)));
            // Масштабируем так, чтобы ширина равнялась 2*PI.

            float fScale = 2 * (float) Math.PI / rectf.Width;
            path.Transform(new Matrix(fScale, 0, 0, fScale, 0, 0));

            // Изменяем контур.

            PointF[] aptf = path.PathPoints;

            for (int i = 0; i < aptf.Length; i++)
                aptfEi] = new PointF(
                    fRadius - (1 + aptf[i].Y) * (float) Math.Cos(aptf[i].X),
                    fRadius - (1 + aptf[i].Y) * (float) Math.Sin(aptf[i].X));

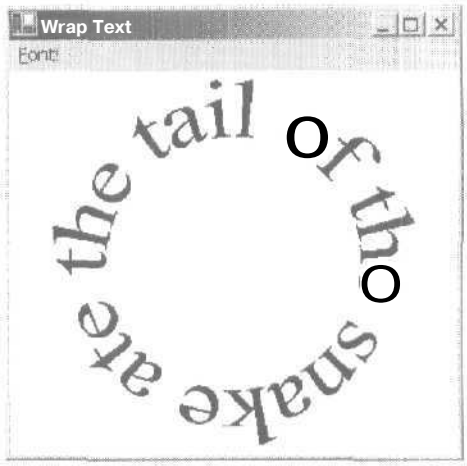
            path = new GraphicsPath(aptf, path.PathTypes);

            // Заполняем контур,

            grfx.TranslateTransform(cx / 2, cy / 2);
            grfx.FillPath(new SolidBrush(clr), path);
        \
}
```

Заметьте: текстовая строка «закольцована». Она не имеет пробелов в начале и в конце, потому что пробелы не стали бы частью контура и последний непробельный символ наложился бы на первый.

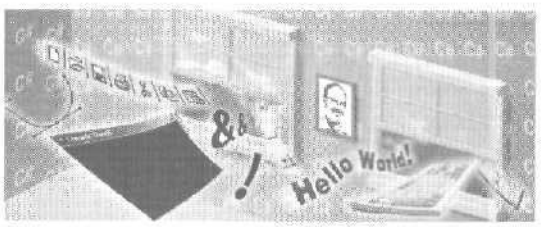
В этой программе начальное матричное преобразование контура сыграло решающую роль в том, чтобы сделать формулы преобразования такими простыми, какие они есть. Контуру задается начальная точка на левой стороне базовой линии шрифта текста, и затем контур масштабируется так, что он имеет шир] ину 2π. Координата *X* контура, таким образом, — просто угол в радианах, и она может быть передана прямо методам *Cos* и *Sin*. Результаты методов *Cos* и *Sin* масштабируются по координате *Y* в сочетании с требуемым радиусом окружности:



Приглядевшись, вы увидите, что базовая линия каждого знака изогнута дугой и что каждый знак в верхней части шире, чем в нижней. Используйте короткую текстовую строку, чтобы этот эффект проявился еще больше.

ГЛАВА

20



Панели инструментов и строки состояния

В пользовательском интерфейсе панели инструментов и строки состояния по распространенности следуют за меню и диалоговыми окнами. Хотя эти два элемента управления выполняют разные функции — панель инструментов обычно состоит из графических кнопок, часто дублирующих команды меню, тогда как строка состояния сообщает пользователю некоторую информацию в текстовом виде, — они во многом схожи. Классы *ToolBar* и *StatusBar* являются потомками класса *Control*, и оба элемента управления размещаются по краям клиентской области. Обычно панель инструментов помещается вверху, а строка состояния — внизу клиентской области. И панель инструментов, и строка состояния объединяют в себе более мелкие элементы. Первая состоит из множества элементов типа *ToolBarButton*, вторая обычно содержит несколько панелей, являющихся экземплярами класса *StatusBarPanel*. И *ToolBarButton*, и *StatusBarPanel* наследуют классу *Component*.

Как вы помните, при добавлении в программу меню строка меню не размещается в клиентской области формы. Вместо этого клиентская область уменьшается в размерах на высоту строки меню. Подобным образом при включении автопрокрутки клиентская область формы уменьшается на размер полос прокрутки.

Однако в этом панели инструментов и строки состояния отличаются от меню и полос прокрутки. В плане «архитектуры» формы они выглядят похоже, но на самом деле являются элементами управления вроде кнопок или текстовых полей. Они обладают свойством *Parent* и располагаются поверх родительских объектов. Когда в качестве контейнера для панели инструментов или строки состояния выступает форма, они располагаются поверх клиентской области. При этом клиентская часть *не* уменьшается в размерах. (Это по большей части историческое различие. Меню и полосы прокрутки существовали с самых ранних версий Windows

и поэтому рассматривались как часть стандартного окна приложения. Панели инструментов и строки состояния появились позже и не считались стандартными элементами.)

Таким образом, добавляя на форму панель инструментов или строку состояния, вы теряете возможность выводить информацию по всей высоте клиентской области. Панель инструментов закрывает верхнюю часть клиентской области, а строка состояния — нижнюю. Однако решается эта проблема довольно легко. Вместо клиентской области весь вывод идет на элемент типа *Panel,* занимающий область между панелью инструментов и строкой состояния.

Реальные приложения, содержащие панель инструментов или строку состояния, должны предоставлять пользователю возможность скрывать эти элементы. Пользователь должен сам решать, стоят ли они той площади, которую занимают. Соответствующую функциональность можно реализовать в меню View (Вид). Спрятать/показать панель инструментов или строку состояния можно, задав их свойству *Visible* значения *false* или *(rue.*

Строка состояния попроще, так что давайте начнем с нее.

Простая строка состояния

Если ваша строка состояния должна выполнять всего одну функцию, скажем, выводить описания элементов меню, то для ее создания вам потребуется ввести в программу лишь 3 строки:

SimpleStatusBar.cs

```
//--------------------------------------------
// SimpleStatusBar.cs f 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
«sing System.Windows.Forms;

class SimpleStatusBar: Form
{
     public static void Main()
     {
          Application.BunCnewSimpleStatusBar());
     }
     public SimpleStatusBar()
     {
          Text = "Simple Status Bar";
          ResizeRedraw = true;

               // Создание строки состояния.

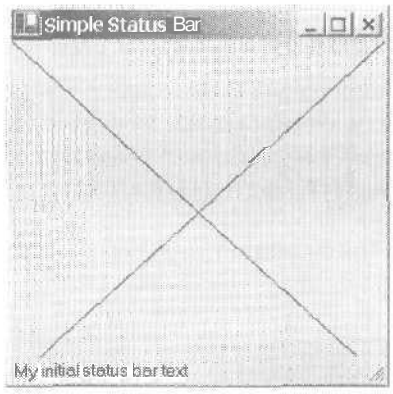
          StatusBar sb = new Status8ar[);
          sb.Parent = this;
          sb.Text - "Hy initial status bar text";
     }
```

см. след. стр.

```
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        Pen      pen  = new Pen(ForeColor);

        grfx.DrawLine(pen, 0, 0, ClientSize.Width, ClientSize.Height);
        grfx.DrawLine(pen, ClientSize.Width, 0, 0, ClientSize.Height);
    }
}
```

Разумеется, в реальной программе вы не стали бы хранить объект *StatusBar* в локальной переменной конструктора, так как вам потребовался бы доступ к этому объекту при обработке событий. Метод *OnPaint* я ввел лишь затем, чтобы показать, что строка состояния скрывает нижнюю часть клиентской области формы:



Заметьте: диагональные линии не доходят до углов формы. На самом деле они прорисованы до самых углов, но скрыты строкой состояния.

Размещение и размер элемента управления *StatusBar* не требуется указывать в явном виде: он располагается снизу формы по всей ее ширине и изменяет свои размеры вместе с формой. Строка состояния ведет себя так вследствие того, что свойство *Dock* объекта типа *StatusBar* по умолчанию инициализируется значением *DockStyle.Bottom*. Справа на строке состояния расположен манипулятор размера. Этот декоративный элемент обозначает область, за которую можно ухватиться при изменении размера окна. Его можно скрыть, установив свойству *SizingGrip* значение *false;*

Свойства *StatusBar* (выборочно)

Тип	Имя	Доступ
bool	*SizingGrip*	Чтение/запись

SizingGrip — это одно из немногих свойств, добавленных к классу *StatusBar*; большинство его свойств унаследовано от класса *Control*.

Свойства *BackColor* и *ForeColor* инициализируются значениями *SystemColors.Control* и *SystemColors.ControlText* и не могут быть изменены. (Но вы вправе изменять свойство *BackColor* формы, тем самым выделяя строку состояния.) *BackgroundImage*

не влияет на вид строки состояния. У строки состояния нельзя изменить и стиль границы,

Свойство *TabStop* инициализируется значением *false,* и его менять вам скорее всего не придется. Если в клиентской области формы расположены другие элементы управления, то вы, вероятно, не захотите, чтобы при нажатии клавиши Tab строке состояния передавался фокус ввода.

Свойства *StatusBar* (выборочно)

Тип	Имя	Доступ
bool	*TabStop*	Чтение/запись

Если в строке состояния должны располагаться несколько элементов, вам потребуются *панели* строки состояния. Параметров у них гораздо большие, чем у самой строки состояния. Вскоре я расскажу о них.

Строка состояния и автоматическая прокрутка

Я уже предупреждал вас о влиянии строки состояния на вид клиентской области. Позвольте дать вам совет: не используйте механизм автопрокрутки на форме совместно со строкой состояния. При этом последняя рассматривается как любой другой элемент управления, расположенный в нижней части формы, и прокручивается с остальной областью.

Чтобы продемонстрировать эту проблему, в приведенной ниже программе свойству *AutoScroll* задано значение *true,*

StatusBarAndAutoScroll.cs

```
//-----------------------------------------------------
// StatusBarAndAutoScroll.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class StatusBarAndAutoScroll: Form
{
    public static void Main()
    {
        Application.Run(new StatusBarAndAutoScroll());
    }
    public StatusBarAndAutoScroll()
    {
        Text = "Status Bar and Auto-Scroll";
        AutoScroll = true;

            // Создание строки состояния.

        StatusBar sb = new StatusBar();
        sb.Parent = this;
```

см. след. стр.

```
            sb.Text = "My initial status bar text";

                // Создание надписей как дочерних объектов формы.

            Label label = new Label();
            label.Parent - this;
            label.Text = "Upper left";
            label.Location = new Point(0, 0);

            label = new Label();
            label.Parent = this;
            label.Text = "Lower right";
            label.Location = new Point(250, 250);
            label.AutoSize * true;
        }
    }
```

Чтобы после запуска программы увидеть строку состояния, вам придется прокрутить клиентскую область вниз! Вот как будет выглядеть окно программы, если прокрутить клиентскую область в нижний правый угол:



Совершенно неприемлемо! Полосы прокрутки не должны влиять на положение строки состояния. Последняя вообще должна находиться за пределами области, окруженной полосами прокрутки. Она должна располагаться *под* горизонтальной полосой прокрутки.

Чтобы эта программа выглядела и работала должным образом (и для решения проблемы вывода информации в клиентскую область), можно задействовать элемент типа *Panel.* Если имеется панель инструментов или строка состояния, программа почти всегда должна начинаться с создания панели или другого элемента управления, заполняющего клиентскую область,

Объект типа *Panel* заполнит клиентскую область, если его свойству *Dock* задать *DockStyle.Fill.* После этого все, что вы хотели сделать в клиентской области (располагать элементы управления или выводить информацию), можно делать на созданной панели. Единственными элементами, контейнером которых является не панель, а форма, остаются панель инструментов и строка состояния.

Вот пример такого решения.

SimpleStatusBarWithPanel.cs

```
//-------------------------------------------------------
// SimpleStatusBarWithPanel.cs © 2001 by Charles Petzold
//-------------------------------------------------------
using System;
«sing System.Drawing;
using System.Windows.Forms;

class SimpleStatusBarWithPanel: Form
{
     public static void Main()
     I
          Application.Run(new SimpleStatusBarWithPanel());
     }
     public SimpleStatusBarWithPanel()
     \
          Text = "Simple Status Bar with Panel";

               // Создание панели.

          Panel panel = new Panel();
          panel.Parent - this;
          panel.BackColor = SystemColors.Window;
          panel.ForeColor = SystemColors.WindowText;
          panel.AutoScroll = true;
          panel.Dock = DockStyle.Fill;
          panel.BorderStyle = BorderStyle.Fixed3D;

               // Создание строки состояния на форме.

          StatusBar sb = new StatusBar();
          sb.Parent * this;
          sb.Text = "My initial status bar text";

               // Создание надписей на панели.

          Label label '" new Label();
          label.Parent = panel;
          label.Text = "Upper left";
          label.Location = new Point(0, 0);

          label - new Label0;
          label.Parent = panel;
          label.Text = "tower right";
          label.Location = new Point(250, 250);
          label.AutoSize = true;
     }
}
```

Заметьте: в программе свойству *AutoScroll* объекта *Panel* присвоено *true,* л на панели созданы два объекта типа *Label.*

Свойству *Dock* панели задано *DockStyle.Fill* поэтому панель принимает размеры клиентской области. Когда затем создается объект *StatusBar,* его свойство *Dock* получает значение *DockStyle.Bottom.* По сути дела объекты *Panel* и *StatusBar* делят клиентскую часть формы на две непересекающиеся области. Если свойству *AutoScroll* панели задать *true,* строка состояния будет расположена под полосами прокрутки:



Панели строки состояния

До сих пор мы использовали строку состояния просто как надпись, расположенную в нижней части формы. Если скрыть манипулятор размера, ее будет трудно отличить от простой надписи.

Объекты типа *StatusBarPanel* позволяют сделать строку состояния более полезной. Строка состояния может содержать несколько таких объектов или не содержать их вовсе. Класс *StatusBar* обладает двумя свойствами, связанными с объектами *StatusBarPanel.*

Свойства *StatusBar* (выборочно)

Тип	Имя	Доступ
bool	*ShowPanels*	Чтение/запись
StatusBar.StatusBarPanelCollection	*Panels*	Чтение

ShowPanels инициализируется значением *false,* и все панели строки состояния игнорируются. На панели отображается лишь текстовая строка из свойства *Text* объекта *StatusBar.* Если задать *ShowPanels* значение *true,* то игнорируется свойство *Text* объекта *StatusBar* — вместо него отображаются текстовые строки объектов *StatusBarPanel* Каждой панели строки состояния можно поставить в соответствие текстовую строку и значок или задействовать механизм прорисовки владельцем класса *StatusBar.*

Свойство *Panels* класса *StatusBar* представляет собой класс-набор, содержащий методы индексирования и управления набором объектов *StatusBarPanel,* такие как *Add, AddRange, Insert* и *Remove.*

Вот простой пример создания строки состояния с двумя панелями.

TwoStatusBarPanels.cs

```
//-------------------------------------------------
// TwoStatusBarPanels.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoStatusBarPanels: Form
{
     public static void Main()
     {
          Application.Run(new  TwoStatysBarPanels0);
     }
     public TwoStatusBarPanels()
     {
          Text = "Two Status flar Panels";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;

          StatusBar sb * new StatusBar();
          sb. Parent = this;
          sb.ShowPanels = true;

          StatusBarPanel sbpanel1 = new StatusBarPanel();
          sbpanel1.Text = "Panel 1";

          StatusBarPanel sbpanel2 = new StatusBarPanel();
          sbpanel2.Text - "Panel 2";
          sb.Panels.Add(sbpanel1);
          sb.Panels.Add(sbpanel2);
     }
}
```

В этой программе создается один объект *StatusBar* и два объекта *StatusBarPanel*. На форме это выглядит так:



Класс *StatusBarPanel* не является потомком класса *Control* и не содержит свойств *BackColor*, *ForeColor*, *Font*, *Location* или *Size*. (Однако содержит свойство *Width*, которое по умолчанию устанавливает ширину панели равной 100 пикселам.) Объект *StatusBarPanel* содержит свойство *Parent*, но оно доступно только для чтения

Сопоставить объект *StatusBarPanel* объекту *StatusBar* можно только через свойство *Panels* объекта *StatusBar*. В программе TwoStatusBarPanels показан стандартный способ такого сопоставления, состоящий в вызове метода *Add* набора *StatusBar.StatusBarPanelCollection*. Другой способ — создать массив объектов *StatusBarPanel* и вызвать метод *AddRange*:

```
sb.Panels.AddRange(new StatusBarPanel[] { sbpanel1, sbpanel2 });
```

Кроме того, можно не создавать объекты *StatusBarPanel* явно, а применить перегруженный метод *Add*, которому требуется передать лишь текстовую строку:

```
sb.Panels.Add("Panel 1");
sb.Panels.Add("Panel 2");
```

Если вам нужен доступ к определенному объекту *StatusBarPanel*, чтобы, скажем, изменить значение его свойства *Text*, можно просто индексировать свойство *Panels* как массив:

```
sb.Panels[1].Text = "New panel 2 text";
```

Свойства класса *StatusBarPanel*

Класс *StatusBarPanel* содержит 10 неунаследованных свойств. Единственное свойство, доступное только для чтения, указывает на объект *StatusBar*, которому сопоставлен данный объект *StatusBarPanel*:

Свойства *StatusBarPanel* (выборочно)

Тип	Имя	Доступ
StatusBar	*Parent*	Чтение

Следующие свойства влияют на отображение текста на панели:

Свойства *StatusBarPanel* (выборочно)

Тип	Имя	Доступ
StatusBarPanelStyle	*Style*	Чтение/запись
string	*Text*	Чтение/запись
Icon	*Icon*	Чтение/запись
string	*ToolTipText*	Чтение/запись

Тип *StatusBarPanelStyle* — это перечисление, указывающее, должна ли панель рассматриваться как область для прорисовки владельцем:

Перечисление *StatusBarPanelStyle*

Член	Значение
Text	1
OwnerDraw	2

Класс *StatusBarPanel* не содержит собственных событий. Событию *DrawItem* класса *StatusBar* в качестве аргумента передается объект *StatusBarPanel* для вывода графической информации.

Чаще всего, конечно, используется свойство *Text*. Слева от текста на панели может отображаться небольшой значок, размер которого можно подобрать экспериментальным путем.

Весьма удобно свойство *ToolTipText*. В нем вы можете указать короткое описание, которое будет появляться при наведении курсора мыши на панель. В следующей программе я покажу пример использования этого свойства.

На вид и размер панели влияют такие свойства:

Свойства *StatusBarPanel* (выборочно)

Тип	Имя	Доступ
StatusBarPanelBorderStyle	*BorderStyle*	Чтение/запись
HorizontalAlignment	*Alignment*	Чтение/запись
StatusBarPanelAutoSize	*AutoSize*	Чтение/запись
int	*Width*	Чтение/запись
int	*MinWidth*	Чтение/запись

Стиль границы панели определяется членами перечисления;

Перечисление *StatusBarPanelBorderStyle*

Член	Значение
None	1
Raised	*2*
Sunken	3

По умолчанию устанавливается значение *Sunken* (панель выглядит утопленной в строку состояния).

Свойство *Alignment* влияет на выравнивание текста внутри панели, но не на положение самой панели. Ему присваиваются значения из перечисления *HorizontalAlignment,* которое также используется другими элементами управления:

Перечисление *HorizontalAlignment*

Член	Значение
Left	0
Right	I
Center	*2*

По умолчанию ширина панели определяется свойством *Width*, которое инициализируется значением 100 пикселов. Свойство *AutoSize* позволяет установить размер панели в соответствии с отображаемым на ней текстом или количеством свободного места:

Перечисление *StatusBarPanelAutoSize*

Член	Значение	Комментарии
None	1	Используется по умолчанию; размер панели регулируется свойством *Width*.
Spring	2	Панель занимает все свободное место.
Contents	3	Ширина панели выравнивается по выводимому тексту.

Свойство *MinWidth* ограничивает минимальную ширину панели и влияет только на панели, свойству *AutoSize* которых задано значение *StatusBarPanelAutoSize.Spring*. По умолчанию свойству *MinWidth* присваивается значение 10 пикселов.

Обычно первой панели строки состояния (расположенной слева) задают стиль границы *StatusBarPanelBorderStyle.None*, а свойству *AutoSize* — *StatusBarPanelAutoSizeSpring*. как правило, ее используют для вывода подсказок к командам меню. Остальным панелям задают стиль границы *StatusBarPanelBorderStyle.Sunken*, а *свойству AutoSize* — *StatusBarPanelAutoSize.None* или *StatusBarPanelAutoSize.Contents*.

Ниже показан пример программы, где в строке состояния создаются три панели в соответствии с приведенным описанием. Две правые панели строки состояния отображают текущую дату и время.

DateAndTimeStatus.cs

```
//-------------------------------------------------
// DateAndTimeStatus.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DateAndTimeStatus: Form
{
     StatusBarPanel sbpMenu, sbpDate, sbpTime;

     public static void Main()
     {
          Application.Run(new DateAndTimeStatus());
     }
     public DateAndTimeStatus()
     {
          Text = "Date and Time Status";
          BackColor * SystemColors.Window;
          ForeColor = SystemColors.WindowText;

               // Создание строки состояния.

          StatusBar sb = new StatusBar();
          sb.Parent = this;
          sb.ShowPanels = true;

               // Создание панелей для строки состояния.
```

см. след. стр.

```
        sbpMenu * new StatusBarPanel();
        sbpMenu.Text = "Reserved for menu help";
        sbpMenu.BorderStyle - StatusBarPanelBorderStyle.None;
        sbpMenu.AutoSize - StatusBarPanelAutoSize.Spring;

        sbpDate = new StatusBarPanel();
        sbpDate.AutoSize = StatusBarPanelAutoSize.Contents;
        sbpDate.ToolTipText = "The current date";

        sbpTime = new StatusBarPanel();
        sbpTime.AutoSize = StatusBarPanelAutoSize.Contents;
        sbpTime.ToolTipText = "The current time";

            // Сопоставление панелей строке состояния.

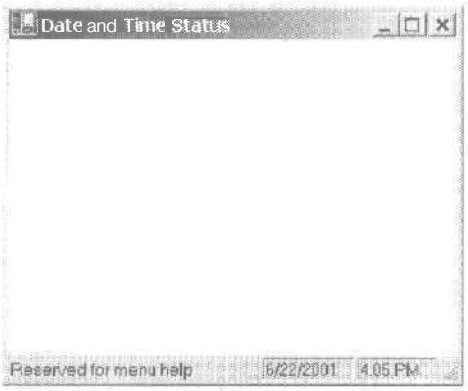
        sb.Panels.AddRange(new StatusBarPanel[]
                                    { sbpMenu, sbpDate, sbpTime });

            // Установка таймера на срабатывание каждую секунду.

        Timer timer = new Timer();
        timer.Tick += new EventHandler(TimerOnTick);
        timer.Interval = 1000;
        timer.Start();
    }
    void TimerOnTick(object obj, EventArgs ea)
    {
        DateTime dt = DateTime.Now;

        sbpDate.Text - dt.ToShortDateString();
        sbpTime.Text - dt.ToShortTimeString();
    }
}
```

Для двух панелей в программе указывается значение свойства *ToolTipText*. Эта строка состояния будет выглядеть так: .



Подсказки для меню

Одна из основных задач строки состояния — отображение *подсказок для меню*. В строке состояния выводится короткий текст, описывающий команды меню, на которые пользователь наводит курсор.

Такие подсказки требуют применения нескольких особенностей меню и форм, не используемых обычно. Как вы помните, класс *MenuItem* поддерживает 5 событий. До сих пор я применял события *Popup* для отображения всплывающего меню и *Click* — при выборе пользователем одного из элементов. Я также приводил пример применения событий *MeasureItem* и *DrawItem* для прорисовки владельцем элементов меню.

Пятое событие класса *MenuItem* — *Select* — возникает при наведении курсора на элемент меню или при перемещении по меню с помощью клавиш-стрелок. Для отображения подсказок к меню следует обрабатывать именно событие *Select,* тогда при перемещении курсора по элементам меню вы сможете выводить в строку состояния текст подсказки для каждого элемента,

При первой попытке добавить подсказки с помощью обработчика события *Select* вы столкнетесь с раздражающей проблемой: последняя подсказка остается в строке состояния и после того, как пользователь щелкнул элемент меню! Можно попробовать разрешить ситуацию, очищая строку состояния в каждом из обработчиков события *Click* для объектов *MenuItem*. А если пользователь закроет меню клавишей Esc? Подсказка снова останется в строке состояния!

И все-таки у этой проблемы *есть* решение. Класс *Form* содержит два события, позволяющие инициализировать и очистить строки состояния:

События класса *Form* (выборочно)

Событие	Метод	Делегат	Аргумент
MenuStart	*OnMenuStart*	*EventHandler*	*EventArgs*
MenuComplete	*OnMenuComplete*	*EventHandler*	*EventArgs*

Эти события возникают в начале и в конце работы с меню, Самое важное событие — *MenuComplete.* При его возникновении следует сбрасывать текст подсказки. При этом в строку состояния можно вернуть текст, который отображался там при возникновении события *MenuStart.*

Следующая программа создает строку состояния с единственной панелью и выводит на нее подсказки для команд меню. При инициализации в строку состояния выводится текст «Ready». Метод *OnMenuStart* сохраняет этот текст, а *OnMenuComplete* снова выводит его в строку состояния.

Для всех элементов меню программа устанавливает один обработчик события *Select* — метод *MenuOnSelect.* Но поскольку программа MenuHelpFirstTry — это просто пример, она не содержит обработчиков событий *Click* ни для одной из команд меню.

MenuHelpFirstTry.cs

```
//-----------------------------------------
// MenuHelpFirstTry.cs © 2001 by Charles Petzold
//.....--------.....--------.....------
using System;
```

см. след. < тр.

```
using System.Drawing;
using System.Windows.Forms;

class MenuHelpFirstTry: Form
{
     StatusBarPanel sbpMenuHe1p;
     string         strSavePanelText;

     public static void Main()
     1
          Application.Run(new MenuHelpFirstTry());
     }
     public MenuHelpFirstTry() ,j
     {
          Text = "Menu Help (First Try)";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;

               // Создание строки состояния с одной панелью.

          StatusBar sb = new StatusBar();
          sb.Parent = this;
          sb.ShowPanels = true;

          sbpMenuHelp = new StatusBarPanel();
          sbpMenuHelp.Text = "Ready";
          sbpMenuHelp.AutoSize * StatusBarPanelAutoSize.Spring;

          sb.Panels.Add(sbpMenuHelp);

               // Построение простого меню.
               // $ данном примере мы можем не вводить обработчики Click,
               // но нам не обойтись без обработчиков событий Select.

          Menu = new MainMenu();
          EventHandler ehSelect = new EventHandler(MenuOnSelect);

               // Создание элементов меню File.

          MenuItem mi = new MenuItem("File");
          mi.Select += ehSelect;
          Menu.MenuItems.Add(mi);

          ffli - new MenuItem("Open");
          mi.Select += ehSelect;
          Menu.MenuItems[0].tenultens.Add(mi);

          mi = new MenuItem("Close");
          mi.Select += ehSelect;
          Menu.MenuItems[0].MenuItems.Add(mi);
```

см. след. стр.

```
        mi * new MenuItem("Save");
        mi.Select += ehSelect;
        Menu.MenuItems[0].MenuItems.Add(mi);

             // Создание элементов меню Edit,

        mi = new MenuItem("Edit");
        mi.Select += ehSelect;
        Menu.MenuItems.Add(mi);

        ml - new MenuItem("Cut");
        mi.Select += ehSelect;
        Menu. MenuItems[ 1].MenuItems.Add(mi);

        mi = new MenuItem("Copy");
        mi.Select += ehSelect;
        Menu.MenuItems[1].MenuItems.Add(mi);

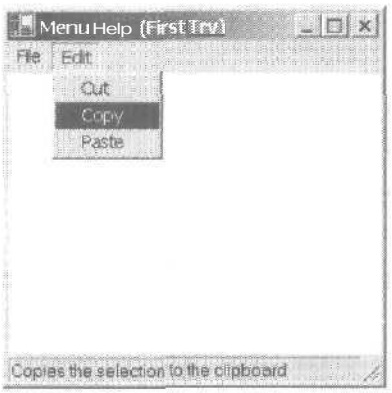
        mi = new MenuItem("Paste");
        mi.Select += ehSelect;
        Menu.MenuItems[1].MenuItems.Add(mi);
    }
    protected override void OnMenuStart(EventArgs ea)
    {
        StrSavePanelText = sbpMenuHelp.Text;
    }
    protected override void OnMenuComplete(EventArgs ea)
    {
        sbpMenuHelp.Text = strSavePanelText;
    }
    void MenuOnSelect(object obj, EventArgs ea)
    {
        MenuItem mi = (MenuItem) obj;
        string   str;

        switch (mi.Text)
        {
        case "File":   str = "Commands for working with files";    break;
        case "Open":   str = "Opens an existing document";         break;
        case "Close":  str = "Closes the current document";        break;
        case "Save":   str = "Saves the current document";         break;
        case "Edit":   str - "Commands for editing the document";  break;
        case "Cut":    str = "Deletes the selection and " +
                             "copies it to the clipboard";         break;
        case "Copy":   str = "Copies the selection to the " +
                             "clipboard";                          break;
        case "Paste":  str = "Replaces the current selection " +
                             "with the clipboard contents";        break;
        default:       str = "";                                   break;
        }
```

см. след. стр.

```
            sbpMenuHelp.Text = str;
        }
    }
```

Метод *MenuOnSelect* определяет название выделенного элемента меню и выбирает соответствующую подсказку, которая затем используется, чтобы установить свойство *Text* объекта *StatusBarPanel*. Вот как выглядит эта программа:



События *Select* возникают независимо от того, доступен ли данный элемент меню, так что такая программа будет отображать подсказку даже для недоступных команд. Иногда пользователей беспокоят недоступные команды меню. (Меня, например, беспокоят.) Текст подсказки может объяснять причину, по которой команда в данный момент недоступна.

Программа MenuHelpFirstTry хорошо работает, и вам может прийтись по вкусу идея объединения всех подсказок в одном месте, таком как метод *MenuOnSelect,* Я-то отношусь к ней без особого энтузиазма: я бы предпочел связать каждую подсказку со своим элементом меню. Но и без этого *MenuOnSelect* не лучшим образом определяет выделенный элемент меню. Конструкция *switch case*, использующая свойство *Text* объекта *MenuItem*, требует точного соответствия текстовых строк, что может создать определенные трудности.

По-моему, изящным решением было бы описать новый класс (скажем, *MenuItemHelp*), подклассом которого сделать класс *MenuItem*. Класс *MenuItemHelp* просто вводил бы новое свойство с именем *HelpText*, в котором можно было бы хранить строку подсказки. Свойство *HelpText* можно устанавливать при создании объекта *MenuItemHelp:*

```
mi.Text = "&Open...";
mi.HelpText = "Opens an existing document";
```

В таком случае метод *MenuOnSelect* становится гораздо проще:

```
void MenuOnSelect(object obj, EventArgs ea)
{
    sbpMenuHelp.Text = ((MenuItemHelp)obj).HelpText;
}
```

Можно пойти дальше, введя в класс *MenuItemHelp* еще одно свойство со ссылкой на панель строки состояния, в которой будет отображаться подсказка. Сам класс

может переопределить метод *OnSelect* для вывода подсказки на панель строки состояния. Файл MenuItemHelp.cs описывает такой класс, наследующий класс *MenuItem*.

MenuItemHelp.cs

```
//---------------------------------------
// MenuItemHelp.cs © 2001 by Charles Petzold
// -------------- ..... -------- ..... ----------
using System;
using System.Drawing;
using System.Windows.Forms;

class MenuItemHelp: MenuItem
{
                                                    // Закрытые поля
    StatusBarPanel sbpHelpPanel;
    string         strHelpText;
                                                    // Конструкторы
    public MenuItemHelp()
    {
    }
    public MenuItemHelp(string strText): base(strText)
    {
    }
                                                    // Свойства
    public StatusBarPanel HelpPanel
    {
        get { return sbpHelpPanel; >
        set { sbpHelpPanel = value; }
    }
    public string HelpText
    {
        get { return strHelpText; }
        set { strHelpText » value; }
    }
                                                    // Переопределение метода
  .' protected override void OnSelect(EventArgs ea)
    {
        base.OnSelect(ea);

        if (HelpPanel I* null)
            HelpPanel.Text = HelpText;
    }
}
```

Ниже приведена версия программы MenuHelpFirstTry, создающая вместо объектов *MenuItem* объекты *MenuItemHelp*. При создании каждого объекта *MenuItemHelp* устанавливаются его свойства *HelpPanel* и *HelpText*.

MenuHelpSubclass.cs

```
//--
// MenuHelpSubclass.cs © 2001 by Charles Petzold
//----
using System;
using System.Drawing;
using System.Windows.Forms;

class MenuHelpSubclass: Form
{
    StatusBarPanelsbpMenuHelp;
    string         strSavePanelText;

    public static void Main()
    {
        Application.Run(new MenuHelpSubclass());
    }
    public MenuHelpSubclassO
    {
        Text * "Menu Help ";
        BackColor = SystemColors.Window;
        ForeColor * SystemColors.WindowText;

            // Создание строки состояния с одной панелью.

        StatusBar sb = new StatusBar();
        sb.Parent = this;
        sb.ShowPanels * true;

        sbpMenuHelp = new StatusBarPanel();
        sbpMenuHelp.Text =  "Ready";
        sbpMenuHelp.AutoSize = StatusBarPanelAutoSize.Spring;

        sb.Panels.Add(sbpMenuHelp);

            // Построение простого меню из элементов MenuItemHelp.

        Menu = new MainMenu();

            /,/ Создание элементов меню File.

        MenuItemHelp mi = new MenuItemHelp("&File");
        mi.HelpPanel = sbpMenuHelp;
        mi.HelpText * "Commands for working with files";
        Menu.MenuItems.Add(mi);

        ad * new MenuItemHelp("&Open...");
        mi.HelpPanel = sbpMenuHelp;
        mi.HelpText = "Opens an existing document";
```

см. след. стр.

```
            Menu.MenuItems[0].MenuItems.Add(mi);

            mi = new MenuItemHelp("&Close");
            ffll.HelpPanel = sbpMenuHelp;
            ffll.WelpText = "Closes the current document";
            Menu.MenuItems[0].MenuItems.Add(mi);

            mi = new MenuItemHelp("&Save");
            mi.HelpPanel= sbpMenuHelp;
            mi.HelpText = "Saves the current document";
            Menu.MenuItems[0].MenuItems.Add(mi);

                // Создание элементов меню Edit.

            mi = new MenuItemHelp("&Edit");
            ffii.HelpPanel = sbpMenuHelp;
            mi.HelpText = "Commands for editing the document";
            Menu.MenuItems.Add(mi);

            mi - new MenuItemHelp("Cu&t");
            mi.HelpPanel = sbpMenuHelp;
            ni.HelpText = "Deletes the selection and " +
                          "copies it to the clipboard";
            Menu.MenuItems[1].MenuItems.Add(mi);

            mi = new MenuItemHelp("&Copy");
            mi.HelpPanel = sbpHenuHelp;
            mi.HelpText = "Copies the selection to the clipboard";
            Menu.MenuItems[1].MenuItems.Add(mi);

            mi - new MenuItemHelp("&Paste");
            mi.HelpPanel = sbpHenuHelp;
            mi.HelpText = "Replaces the current selection " +
                          "with the clipboard contents";
            Menu.MenuItems[1].MenuItems.Add(mi);
        }
        protected override void OnMenuStart(EventArgs ea)
        {
            strSavePanelText = sbpMenuHelp.Text;
        }
        protected override void OnMenuComplete(EventArgs ea)
        {
            sbpMenuHelp.Text = strSavePanelText;
        }
    }
```

Простая панель инструментов

В конце главы 11 я рассказывал о классе *ImageList*, представляющем собой набор изображений одинакового размера и глубины цвета. Элемент управления *ToolBar*

состоит из объекта *ImageList* и набора объектов *ToolBarButton*. На каждой кнопке выводится одно из изображений, хранящихся в объекте *ImageList*.

Большинство приложений использует для кнопок панели инструментов изображения размером 16×16 пикселов. Для объекта *ImageList* этот размер задается по умолчанию, и в этой главе я использую изображения именно такого размера. Если вам нужны кнопки большего или меньшего размера — и даже непрямоугольной формы — прежде всего, очевидно, следует создать растровые изображения нужного размера. Перед внесением изображений в объект *ImageList* задайте свойству *ImageSize* значение, равное размеру изображений. Этот размер будут затем использовать все объекты *ToolBar* и *ToolBarButton*.

Вот важнейшие свойства объекта *ToolBar*:

Свойства *ToolBar* (выборочно)

Тип	Имя	Доступ
ImageList	***ImageList***	Чтение/запись
ToolBar.ToolBarButtonCollection	***Buttons***	Чтение
bool	*ShowToolTips*	Чтение/запись

Каждому объекту *ToolBar* сопоставляется один объект *ImageList,* который обычно содержит несколько изображений.

Свойство *Buttons* представляет собой набор того типа, с которым мы уже сталкивались. Элементы набора *Buttons* можно индексировать как обычный массив; каждый элемент представляет собой объект типа *ToolBarButton*. Класс *ToolBarButtonCollection* содержит несколько методов, в том числе *Add*, *AddRange*, *Insert* и *Remove*, позволяющих управлять кнопками панели инструментов.

Я включил *ShowToolTips* в список важнейших свойств, так как кнопкам панели инструментов довольно легко сопоставить подсказки, но они не будут отображаться, если не присвоить *true* свойству *ShowToolTips*.

Вот важнейшие свойства объекта *ToolBarButton:*

Свойства *ToolBarButton* (выборочно)

Тип	Имя	Доступ
int	*ImageIndex*	Чтение/запись
string	*ToolTipText*	Чтение/запись

Свойство *ImageIndex* указывает на конкретное изображение из объекта *ImageList,* отображаемое на данной кнопке. В свойстве *ToolTipText* хранится текст подсказки, которая появляется при наведении на кнопку курсора. Изображения на кнопках не всегда очевидны, поэтому подсказки могут здорово помочь.

Вскоре я расскажу о событиях объекта *ТооШаг,* но сначала взглянем на программу, которая ничего не делает — только отображает меню, которое ничего не делает, и панель инструментов, которая тоже ничего не делает. Для кнопок панели инструментов используются изображения из файла StandardButtons.bmp.

SimpleToolBar.cs

```
//---------------------------------------------
// SimpleToolBar.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleToolBar: Form
{
     public static void Main()
     {
          Application.Run(new SimpleToolBar());
     }
     public SimpleToolBar()
     {
          Text = "Simple Toolbar";

               // Создание несложного меню (просто для демонстрации).

          Menu = new MainMenu();
          Menu.MenuItems.Add("File");
          Menu.MenuItems.Add("Edit");
          Menu.MenuItems.Add("View");
          Menu.MenuItems.Add("Help");

               // Создание набора изображений.

          Bitmap bm = new Bitmap(GetType(),
                                 "SimpleToolBar.StandardButtons.bmp");

          ImageList imglst - new ImageList();
          imglst.Images.AddStrip(bm);
          imglst.TransparentColor = Color.Cyan;

               // Создание панели инструментов.

          ToolBar tbar = new ToolBar();
          tbar.Parent = this;
          tbar.ImageList = imglst;
          tbar.ShowToolTips = true;

               // Создание кнопок панели инструментов.

          string[] astr = {"New", "Open", "Save", "Print",
                           "Cut", "Copy", "Paste" };

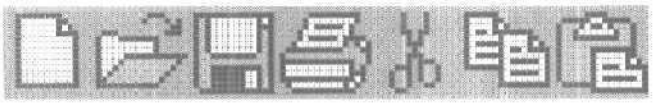
          for (int i = 0; i < I; i++)
          {
```

см. след. стр.

```
            ToolBarButton tbarbtn = new ToolBarButton();
            tbarbtn.ImageIndex = i;
            tbarbtn.ToolTipText = astr[i];

            tbar.Buttons.Add(tbarbtn);
        }
    }
}
```

StandardButtons.bmp



Картинки для кнопок можно создавать по отдельности или совместить несколько изображений в одном растровом файле. В программе SimpleToolBar используется стандартный для объекта *ImageList* размер изображений 16×16 пикселов. Файл StandardButtons.bmp содержит 7 изображений для 7 кнопок, так что общий размер растрового изображения получается 112×16 пикселов. Ширина растрового изображения для объекта *ImageList* должна быть кратна его высоте. Кроме того, как я говорил в главе 11, при включении растрового изображения в проект Visual C# .NET свойству Build Action надо задать значение Embedded Resource.

Если вам не доставляет удовольствия процесс создания картинок для кнопок, можете использовать стандартные изображения, поставляющиеся с Visual Studio .NET. Они находятся в подкаталогах папки \Program Files\Microsoft Visual Studio .NET\Common7\Graphics\Bitmaps. Изображения для кнопок панели инструментов хранятся в подкаталогах OffCtlBr и Assorted.

Программа SimpleToolBar начинается с создания нескольких элементов меню верхнего уровня. Я создал их только затем, чтобы показать как меню и панель инструментов смотрятся вместе. Далее в качестве ресурса загружается изображение из файла StandardButtons.bmp. (Подробнее о загрузке ресурсов см. главу 11.) Программа создает объект *ImageList* и при помощи метода *AddStrip* свойства *Images* вносит изображение в список. Так как свойство *ImageSize* по умолчанию устанавливает размер изображений 16×16 пикселов, объект *ImageList* интерпретирует полученное изображение как набор из 7 картинок.

На черно-белой иллюстрации не видно, что изображение из файла StandardButtons.bmp нарисовано на светло-голубом фоне. В программе для объекта *ImageList* этот цвет определяется как прозрачный.

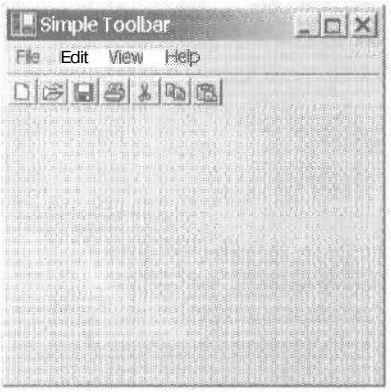
Далее создается сама панель. Для нее устанавливаются три свойства: *Parent* определяет форму в качестве контейнера панели, *ImageList* сопоставляет панели набор изображений для кнопок, а значение *ShowToolTips* разрешает отображение подсказок.

Далее программа создает 7 объектов *ToolBarButton*, устанавливая их свойствам *ImageIndex* значения от 0 до 6 согласно 7 изображениям для этих кнопок. (Если порядок хранения картинок не соответствует порядку кнопок, который вы хотите задать, их можно переупорядочить, выставив *значения ImageIndex*.) Каждой кноп-

ке сопоставляется короткая подсказка. Программа добавляет кнопки на панель методом *Add* свойства *Buttons*:

```
tbar.Buttons.Add(tbarbtn);
```

Вот что мы получили:



Поэкспериментировав с этой программой, вы обнаружите, что кнопки на панели во многом похожи на обычные кнопки, Ниже я расскажу, как создавать двухпозиционные кнопки или кнопки, открывающие меню.

Вероятно, вам захочется как-то разделить панель инструментов и клиентскую часть формы. Скорее всего для вывода информации придется задействовать элемент управления *Panel,* для цвета фона которого можно задать значение *SystemColors.Window* (обычно это белый цвет).

Разновидности панелей инструментов

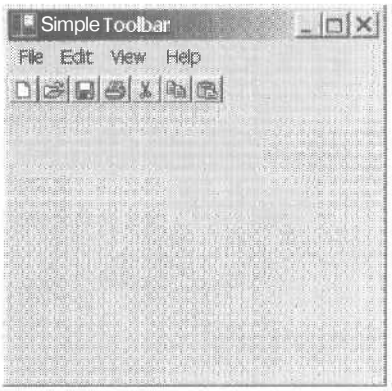
Прежде чем перейти к обработке событий панели инструментов, взглянем на некоторые свойства, влияющие на ее вид:

Свойства *ToolBar* (выборочно)

Тип	Имя	Доступ	Значение по умолчанию
bool	*Wrappable*	Чтение/запись	*true*
bool	*Divider*	Чтение/запись	*true*
BorderStyle	*BorderStyle*	Чтение/запись	*BorderStyle None*
ToolBarAppearance	*Appearance*	Чтение/запись	*ToolBarAppearance.Normal*
ToolBarTextAlign	*TextAlign*	Чтение/запись	*ToolBarTextAlign.Underneath*

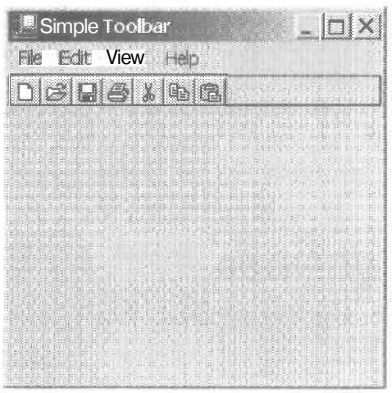
Если ширина формы становится меньше ширины панели инструментов и свойству *Wrappable* задано *true,* панель отображается в две строки. В противном случае панель обрезается по правому краю.

Свойство *Divider* влияет на отображение линии, отделяющей панель от меню. Если задать *ему false,* разделитель исчезает:



По эстетическим соображениям разделитель, наверное, лучше оставить.

Свойство *BorderStyle* (которому можно также задавать значения *FixedSingle* или *Fixed3D*) влияет на отображение границы вокруг панели инструментов. Если отключить разделитель и установить свойству *BorderStyle* значение *FixedSingle,* панель выглядит так:



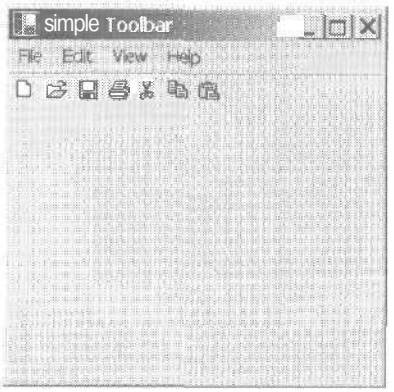
По-моему, разделитель выглядит лучше границ,

Свойство *Appearance* может принимать одно из значений перечисления *ToolBarAppearance*:

Перечисление *ToolBarAppearance*

Член	Значение
Normal	0
Flat	1

Похоже, в современных приложениях предпочтительнее плоская панель инструментов (значение *Flat*). Вот как выглядит плоская панель, если включить разделитель и отключить отображение границы:



Плоская панель выглядит лучше, если пространство под ней окрашено в другой цвет.

Свойство *TextAlign* может принимать значения перечисления *ToolBarTextAlign*:

Перечисление ***ToolBarTextAlign***

Член	Значение
Underneath	0
Right	1

Это свойство касается еще не изученной нами особенности. Дело в том, что кнопки на панели инструментов могут быть подписаны:

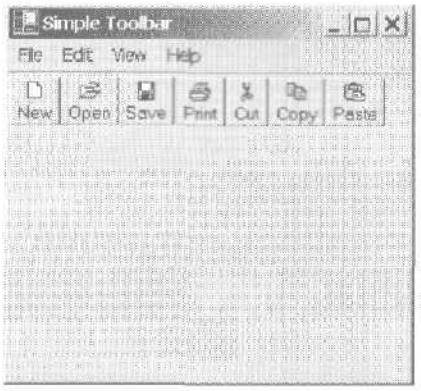
Свойства Too/Bar **(выборочно)**

Тип	Имя	Доступ
string	*Text*	Чтение/запись

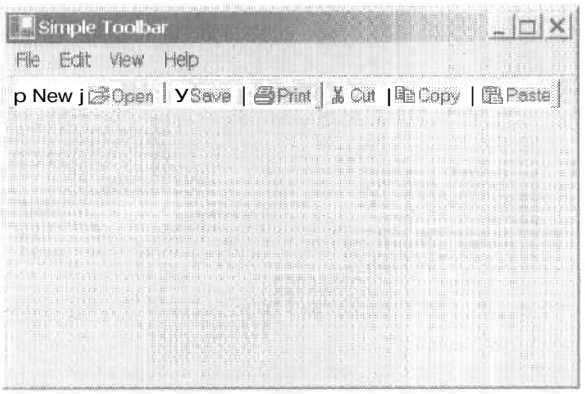
Например, если включить в цикл *for* программы SimpleToolBar оператор:

```
tbarbtn.Text = astr[i];
```

кнопки будут выглядеть вот так:



Заметьте: ширина кнопок теперь непостоянна. Она зависит от длины подписи. *Если* свойству *TextAlign*панели задать значение *ToolBarTextAlignRight*кнопки станут шире, но ниже:



События панели инструментов

Программа SimpleToolBar не содержала обработчиков событий и поэтому не могла реагировать на щелчки кнопок. В дополнение к событиям, наследуемым от класса *Control,* класс *ToolBar*содержит два собственных события:

События *ToolBar*

Событие	Метод	**Делегат**	Аргумент
ButtonClick	*OnButtonClick*	*ToolBarButton ClickEventHandler*	*ToolBarButton ClickEventArgs*
ButtonDropDown	*OnButtonDropDown*	*ToolBarButton ClickEventHandler*	*ToolBarButton CHckEventArgs*

Событие *ButtonClick* является наиболее важным. Оно возникает, когда пользователь щелкает одну из кнопок панели инструментов. Событие *ButtonDropDown* применимо только к кнопкам определенного стиля, использующимся для вызова меню. (Вскоре я расскажу об этих кнопках.)

Заметьте: эти события принадлежат классу панели инструментов, а не отдельным кнопкам. Класс *ToolBarButton* событий не содержит. Таким образом, щелчок любой кнопки панели вызывает один и тот же обработчик события. Поэтому внутри обработчика нужно определить, какую именно кнопку щелкнули. Передающийся обработчику аргумент *ToolBarButtonClickEventArg*содержит единственное свойство, позволяющее это сделать:

Свойство *ToolBarButtonCUckEventArgs*

Тип	Имя	Доступ
ToolBarButton	*Button*	Чтение/запись

Для обработки события *ButtonClick* можно сначала сохранить все объекты *ToolBarButton* в качестве полей, а затем, при обработке события, сравнивать свойство *Button* с этими полями. Другой способ — использовать свойство *Tag* объекта *ToolBar-*

Button для хранения любой информации, по которой можно будет определить нажатую кнопку:

Свойства *ToolBarButton* (выборочно)

Тип	Имя	Доступ
object	*Tag*	Чтение/запись

Часто кнопки панели инструментов дублируют функции меню, поэтому весьма разумно сопоставить свойству *Tag* кнопки объект *MenuItem*, соответствующий дублируемому элементу меню.

Рассмотрим простой и полезный пример. Text Box WithToolBar — версия программы Notepad из главы 18 — упрощена настолько, что содержит только меню Edit с командами Cut, Copy и Paste. Но зато в ней есть панель инструментов, дублирующая эти команды. Изображения для объекта *ToolBar* содержатся в объекте *ImageList*, куда они загружаются из растрового изображения StandardButtons.bmp, которое я использовал в программе SimpleToolBar. Но здесь для команд Cut, Copy и Paste применяются только изображения с номерами 4, 5 и 6.

TextBoxWithToolBar.cs

```
//-------------------------------------------------
// TextBoxWithToolBar.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TextBoxWithToolBar: Form
{
     TextBox       txtbox;
     MenuItem      miEditCut, fciEdltCopy, miEditPaste;
     ToolBarButton tbbCut, tbbCopy, tbbPaste;

     public static void Main()
     {
          // Без этого оператора программа не запустится.

          System.Threading.Thread.CurrentThread.ApartmentState =
                                        System.Threading.ApartmentState.STA;

          Application.Run(new TextBoxWithToolBar());
     }
     public TextBoxWithToolBar()
     {
          Text * "Text Box with Toolbar";

               // Создание текстового поля,

          txtbox          * new TextBox();
```

см, след. стр.

```
txtbox.Parent      = this;
txtbox.Dock        = DockStyle.Fill;
txtbox.Multiline   = true;
txtbox.ScrollBars  = ScrollBars.Both;
txtbox.AcceptsTab  = true;

    // Создание набора изображений.

Bitmap bm = new Bitmap(GetType(),
                       "TextBoxWithToolBar.StandardButtons.bmp");

ImageList imglst = new ImageList();
imglst.Images.AddStrip(bm);
imglst.TransparentColor = Color.Cyan;

    // Создание панели инструментов с обработчиком события
    // ButtonClick.

ToolBar tbar = new ToolBar();
tbar.Parent = this;
tbar.ImageList = imglst;
tbar.ShowToolTips = true;
tbar.ButtonClick +=
            new ToolBarButtonClickEventHandler(ToolBarOnClick);

    // Создание меню Edit.

Menu = new MainMenu();

MenuItem mi = new MenuItem("&Edit");
mi.Popup += new EventHandler(MenuEditOnPopup);
Menu.MenuItems.Add(mi);

    // Создаем в меню Edit команду Cut ...

miEditCut = new MenuItem("Cu&t");
miEditCut.Click += new EventHandler(MenuEditCutOnClick);
miEditCut.Shortcut = Shortcut.CtrlX;
Menu.MenuItems[0].MenuItems.Add(miEditCut);

    // ... и соответствующую кнопку на панели инструментов.

tbbCut = new ToolBarButton();
tbbCut.ImageIndex = 4;
tbbCut.ToolTipText = "Cut";
tbbCut.Tag = miEditCut;
tbar.Buttons.Add(tbbCut);

    // Создаем в меню Edit команду Copy ...
```

см. след. стр.

```
        miEditCopy = new MenuItem("&Copy");
        raiEditCopy.Click += new EventHandler(MenuEditCopyOnClick);
        miEditCopy.Shortcut = Shortcut.CtrlC;
        Menu.MenuItems[0].MenuItems.Add(miEditCopy);

            // И ... и соответствующую кнопку на панели инструментов.

        tbbCopy = new ToolBarButton();
        tbbCopy.ImageIndex = 5;
        tbbCopy.ToolTipText = "Copy";
        tbbCopy.Tag - miEditCopy;
        tbar.Buttons.Add(tbbCopy);

            // Создаем в меню Edit команду Paste ...

        miEditPaste = new MenuItem("&Paste");
        miEditPaste.Click += new EventHandler(MenuEditPasteOnClick);
        miEditPaste.Shortcut = Shortcut.CtrlV;
        Menu.MenuItems[0].MenuItems.Add(miEditPaste);

            // ... и соответствующую кнопку на панели инструментов.

        tbbPaste = new ToolBarButton();
        tbbPaste.ImageIndex - 6;
        tbbPaste.ToolTipText = "Paste";
        tbbPaste.Tag = miEditPaste;
        tbar.Buttons.Add(tbbPaste);

            // Создаем таймер, контролирующий состояние кнопок.

        Timer timer = new Timer();
        timer.Interval - 250;
        timer.Tick += new EventHandler(TimerOnTick);
        timer.Start();
    }
    void MenuEditOnPopup(object obj, EventArgs em)
    {
        miEditCut.Enabled =
        ffliE4itGopy.Enabled = (txtbox.SelectionLength > 0);
        miEditPaste. Enabled =
            Clipboard.GetDataObject().GetDataPresent(typeof(string));
    }
    void TimerOnTick(object obj, EventArgs ea)
    {
        tbbCut.Enabled =
        tbbCopy.Enabled = (txtbox.SelectionLength) > 0;
        tbbPaste.Enabled =
            Clipboard.GetDataObject().GetDataPresent(typeof(string));
    }
    void ToolBarOnClick(object obj, ToolBarButtonClickEventArgs tbbcea)
```

см. след. стр.

```
    {
        ToolBarButton tbb = tbbcea.Button;
        MenuItem rai - (MenuItem) tbb.Tag;

        mi.PerformClick();
    }
    void MenuEditCutOnClick(object obj, EventArgs ea)
    {
        txtbox.Cut();
    }
    void MenuEditCopyOnClick(object obj, EventArgs ea)
    {
        txtbox.Copy();
    }
    void MenuEditPasteOnClick(object obj, EventArgs ea)
    {
        txtbox.Paste();
    }
}
```

При создании объекта *ToolBar* в конструкторе в качестве обработчика события *ButtonClick* устанавливается метод *ToolBarOnClick*:

```
tbar.ButtonClick += new ToolBarButtonClickEventHandler(ToolBarOnClick);
```

После создания каждого элемента меню создается объект *ToolBarButton*, соответствующий данному элементу. Свойство *Tag* каждого объекта *ToolBarButton* ссылается на соответствующий объект *Menultern*. Вот как это выглядит для команды Cut:

```
tbbCut.Tag = miEditCut;
```

В результате такого сопоставления метод *ToolBarOnClick* получается довольно простым:

```
void ToolBarOnClick(object obj, ToolBarButtonClickEventArgs tbbcea)
\
    ToolBarButton tbb = tbbcea.Button;
    MenuItem mi = (MenuItem) tbb.Tag;

    mi.PerformClick();
}
```

Метод получает объект *ToolBarButton* на котором был произведен щелчок, сопоставляет его свойство *Tag* элементу *MenuItem* и имитирует щелчок. Это решение настолько элегантно, что позволяет сократить тело обработчика до одной строки:

```
((MenuItem) tbbcea.Button.Tag).PerformClick();
```

Единственная трудность — перевод кнопок панели инструментов в неактивное состояние, если в данный момент нельзя применить команды Cut, Copy или Paste. Обычно элементы меню Cut, Copy и Paste переводятся в активное/неактивное состояние по событию *Popup* для меню Edit. Но так как кнопки видны всегда, для них потребуется иной способ.

Если бы вместо объекта *TextBox* в качестве текстового поля мы задействовали объект *RichTextBox*, можно было бы ввести обработчик события *SelectionChanged*. (Объект *TextBox* не поддерживает это событие.) Обработчик мог бы активизировать кнопки Cut и Copy только при выделении части текста подобно тому, как это делает обработчик события *Popup* для меню Edit.

Однако это не решило бы проблему активизации кнопки Paste. Она должна быть доступна при наличии текста в буфере обмена. Наверное, единственное решение этой проблемы (исключая перехват сообщений Win32 об изменениях содержимого буфера обмена) — создать объект *Timer* и проверить состояние буфера обмена при обработке события *Tick*. А раз уж пришлось создавать таймер, я решил реализовать в обработчике события *Tick* управление состоянием всех трех кнопок.

Кроме свойства *Enabled*, класс *ToolBarButton* также содержит свойство *Visible*:

Свойства *ToolBarButton* (выборочно)

Тип	Имя	Доступ
bool	*Visible*	Чтение/запись
bool	*Enabled*	Чтение/запись

Стили панели инструментов

До сих пор мы создавали панели инструментов, кнопки на которых работали обычным образом. Но панель также может содержать кнопки, работающие как флажки и способные находиться в нажатом/отжатом состоянии. Или в ответ на щелчок кнопки может отображаться всплывающее меню. Все это позволяют реализовать свойства:

Свойства *ToolBarButton* (выборочно)

Тип	Имя	Доступ
ToolBarButtonStyle	*Style*	Чтение/запись
bool	*Pushed*	Чтение/запись
bool	*PartialPush*	Чтение/запись
Menu	*DropDownMenu*	Чтение/запись

Несмотря на определение, *DropDownMenu* на самом деле является свойством типа *ContextMenu*. Свойство *Style* принимает одно из значений:

Перечисление *ToolBarButtonStyle*

Член	Значение
PushButton	1
ToggleButton	2
Separator	3
DropDownButton	4

По умолчанию используется значение *ToolBarButtonStyle.PushButton*.

Свойства *Pushed* и *PartialPush* применимы только к тем кнопкам, свойству *Style* которых задано значение *ToolBarButtonStyle.ToggleButton*. Такие кнопки могут находиться в нажатом/отжатом состоянии. При каждом щелчке такой кнопки про-

исходит событие *ButtonClick*, а свойство *Pushed* меняет значение на противоположное. Программа может изменять значение свойства *Pushed*, тем самым управляя состоянием кнопки. Установка *true* свойству *PartialPush* переводит кнопку в специальное промежуточное состояние (подобно свойству *ThreeState* элемента управления *CheckBox*, описанного в главе 12).

При использовании стиля *ToolBarButtonStyle.Separator* кнопка игнорирует связанный с ней текст или картинку и отображается на панели инструментов в виде разделителя. Если свойству *Appearance* панели инструментов задать значение *ToolBarAppearance.Flat* разделитель выглядит как вертикальная линия, похожая на ту, что отделяет панель инструментов от строки меню. Если этому свойству задать *ToolBarAppearance.Normal* разделитель выглядит как небольшой промежуток между кнопками.

Если свойству *Style* присвоить *ToolBarButtonStyle.DropDownButton*, то при щелчке данной кнопки будет отображаться всплывающее меню, вид которого определяется свойством *DropDownMenu* кнопки. Если меню перед отображением нужно как-то инициализировать, сделать это можно при обработке события *ButtonDropDown*.

Сначала разберемся с кнопками-переключателями. Следующая программа отображает панель инструментов с четырьмя кнопками, картинки для которых взяты из файла FontStyleButtons.bmp. Кнопки позволяют изменять стиль шрифта отображаемого текста, делая его полужирным, курсивным, подчеркнутым или зачеркнутым.

ToggleButtons.cs

```
//-----------------------------------------------
// ToggleButtons.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ToggleButtons: Form
{
     protected Panel   panel;
     protected ToolBar tbar;
     protected string  strText = "Toggle";
     protected Color   clrText = SystemColors.WindowText;
     FontStyle         fontstyle = FontStyle.Regular;

     public static void Main()
     {
          Application.Run(new ToggleButtons());
     }
     public   ToggleButtons()
     {
          Text = "Toggle Buttons";

               // Создание панели для заполнения клиентской части формы.
```

см. след. стр.

```
        panel - new Panel();
        panel.Parent - this;
        panel.Dock = DockStyle.Fill;
        panel.BackColor * SystemColors.Window;
        panel. ForeColor = SystemColors.WindowText;
        panel.Resize += new EventHandler(PanelOnResize);
        panel.Paint += new PaintEventHandler(PanelOnPaint);

            // Создание списка изображений.

        Bitmap bm = new Bitmap(GetType(),
                               "ToggleButtons.FontStyleButtons.bmp");

        ImageList imglst = new ImageList();
        imglst.ImageSize = new Size(bm.Width / 4» bm.Height);
        imglst.Images.AddStrip(bm);
        imglst.TransparentColor - Color.White;

            // Создание панели инструментов.

        tbar = new ToolBar();
        tbar.ImageList = imglst;
        tbar.Parent = this;
        tbar.ShowToolTips = true;
        tbar.ButtonClick +=
                    new ToolBarButtonClickEventHandler(ToolBarOnClick);

            // Создание кнопок панели.

        FontStyle[] afs = { FontStyle.Bold, FontStyle.Italic,
                            FontStyle.Underline, FontStyle.Strikeout };

        for <lnt i = 0; i < 4; i++)
        {
            ToolBarButton tbarbtn = new ToolBarButton();
            tbarbtn.ImageIndex - i;
            tbarbtn.Style - ToolBarButtonStyle.ToggleButton;
            tbarbtn.ToolTipText = afs[i].ToString();
            tbarbtn.Tag = afs[i];

            tbar.Buttons.Add(tbarbtn);
        }
    }
    void ToolBarOnClick(object obj, ToolBarButtonClickEventArgs tbbcea)
    {
        ToolBarButton tbarbtn = tbbcea.Button;

            // Если значение свойства Tag не равно FontStyle, ничего
            // додать не надо.
```

см. след. стр.

```
            if (tbarbtn.Tag == null ||
                tbarbtn.Tag.GetType() != typeof(FontStyle))
                return;

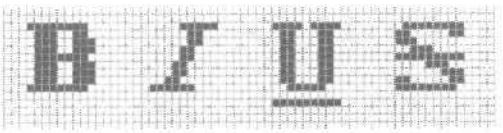
            // Установка или сброс битое поля font sty le.

            if (tbarbtn.Pushed)
                fontstyle |= (FontStyle) tbarbtn.Tag;
            else
                fontstyle 1= "(FontStyle) tbarbtn.Tag;

            panel.Invalidate();
        }
        void PanelOnResize(object obj, EventArgs ea)
        {
            Panel panel = (Panel) obj;
            panel.Invalidate();
        }
        void PanelOnPaint(object obj, PaintEventArgs pea)
        {
            Panel     panel = (Panel) obj;
            Graphics grfx  * pea.Graphics;
            Font      font  = new Font("Times New Roman", 72, fontstyle);
            SizeF     sizef = grfx.MeasureString(strText, font);

            grfx.DrawString(strText, font, new SolidBrush(clrText),
                            (panel.Width - sizef.Width) / 2,
                            (panel.Height - sizef.Height) / 2);
        }
    }
```

FontStyleButtons.bmp



Программа создает элемент управления *Panel* для заполнения клиентской области формы (по крайней мере пока не будет создана панель инструментов) и вывода информации, которая в обычном случае отображалась бы в клиентской области. Обработчик события *PanelOnPaint* выводит в центр панели строку текста. Так как строка всегда отображается посредине, панель нужно перерисовывать при каждом изменении размера. Удобнее всего было бы применить свойство *ResizeRedraw,* но оно является защищенным. Можно создать подкласс для класса *Panel* или (как сделано в этой программе) установить обработчик события *Resize* и делать панель недействительной после каждого изменения размера.

Заметьте: объект *Font,* созданный в методе *PanelOnPaint,* использует аргумент *FontStyle,* хранящийся в битовом поле с *именем fontstyle.* Значения битов этого поля и будут регулироваться при помощи кнопок.

Как и TextBoxWithToolBar, эта программа в качестве обработчика события *ButtonClick* панели устанавливает метод *ToolBarOnClick*. В конструкторе свойству *Style* каждой кнопки присваивается *ToolBarButtonStyle.ToggleButton*, а свойству *Tag* – значение одного из членов перечисления *FontStyle*.

И вновь грамотное использование свойства *Tag* заметно упростило метод *ToolBarOnClick*. После получения объекта *ToolBarButton* в качестве аргумента события и проверки соответствия свойства *Tag* объекту *FontStyle* метод *ToolBarOnClick* устанавливает/сбрасывает нужный бит в поле *fontstyle*.

```
if (tbarbtn.Pushed)
     fontstyle |= (FontStyle) tbarbtn.Tag;
else
     fontstyle &= ~(FontStyle) tbarbtn.Tag;
```

Далее метод делает панель недействительной, что приводит к вызову метода *PanelOnPaint*.



Хотя я приводил примеры кнопок со стилями *PushButton* и *ToggleButton* раздельно, не стоит забывать о том, что любая панель инструментов может содержать кнопки нескольких стилей. Третий из возможных стилей кнопок — *ToolBarButtonStyle.DropDownButton*. При щелчке такой кнопки отображается меню, сопоставленное свойству *DropDownMenu*.

В определении свойства *DropDownMenu* указан тип *Menu*, но на самом деле оно является объектом типа *ContextMenu* (потомком класса *Menu*). Сопоставить свойству *DropDownMenu* нужный объект можно так:

```
ContextMenu menu = new ContextMenu();
menu.MenuItems.Add("First Item");
menu.MenuItems.Add("Second Item");
menu.MenuItems.Add("Third Item");
```

Разумеется, нужно также задать обработчики для каждого элемента меню. После этого соответствующим образом устанавливаются свойства объекта *ToolBarButton*:

```
tbarbtn.Style = ToolBarButtonStyle.DropDownMenu;
tbarbtn.DropDownMenu = menu;
```

Класс *ToolBar* содержит свойство, управляющее выводом небольших стрелок, показывающих, что данная кнопка вызывает всплывающее меню:

Свойства Too/Bar (выборочно)

ТИП	Имя	Дос ı ум
bool	*DropDownArrows*	Чтение/запись

По умолчанию этому свойству присваивается значение *true*.

С кнопками *DropDownMenu* связана небольшая загвоздка, о которой я еще не говорил. Это скорее условность, но если вы видели приложения, использующие такие кнопки, то могли убедиться, что обычно они не вызывают обычные текстовые меню. Меню, вызываемое щелчком такой кнопки, состоит из небольших изображений. Поэтому кнопки типа *DropDownMenu* почти всегда требуют применения меню с прорисовкой владельцем.

Ниже приведена программа DropDownMenuButton, которая добавляет на панель инструментов программы Toggle Buttons пятую кнопку, управляющую цветом текста. Щелчок этой кнопки вызывает меню с прорисовкой владельцем, отображающее стандартные 1б цветов. Кроме того, изображение на кнопке изменяется согласно выбранному цвету.

DropDownMenu Button.cs

```
//-------------------------------------------------------
// DropDownMenuButton.cs © 2001 by Charles Petzold
//-------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DropDownMenuButton: ToggleButtons
{
     public new static void Main()
     {
          Application.Run(new DropDownMenuButton());
     }
     public DropDownMenuButton()
     {
          Text = "Drop-Down Menu Button";
          strText = "Drop-Down";

               // Создаем изображение для новой кнопки и добавляем его
               // в список ImageList.

          tbar.ImageList.Images.Add(CreateBitmapButton(clrText));

               // Создаем меню для кнопки.

          ContextMenu menu = new ContextMenu();

          EventHandler ehOnClick = new EventHandler(MenuColorOnClick);
          MeasureItemEventHandler ehOnMeasureItem =
                    new MeasureItemEventHandler(MenuColorOnMeasureItem);
```

см. след. стр.

```
        DrawItemEventHandler ehOnDrawItem =
                new DrawItemEventHandler(MenuColorOnDrawItem);

        Colort] acolor =
        {
        Color.FromArgb(0x00, 0X00, 0x00), Color.FromArgb(0x00, 0x00, 0x80),
        Color.FromArgb(0x00, 0x80, 0x00), Color,FromArgb(0x00, 0x80, 0x80),
        Color.FromArgb(0x80, 0x00, 0x00), Color.FromArgb(0x80, 0x00, 0x80),
        Color.FromArgb(0x80, 0x80, 0x00), Color.FromArgb(0x80, 0x80, 0x80),
        Color.FromArgb(0xC0, 0xC0, 0xC0), Color.FromArgb(0x00, 0x00, 0xFF),
        Color.FromArgb(0x00, 0xFF, 0x00), Color.FromArgb(0x00, 0xFF, 0xFF),
        Color.FromArgb(0xFF, 0x00, 0x00), Color,FromArgb(0xFF, 0x00, 0xFF),
        Color.FromArgb(0xFF, 0xFF, 0x00), Color,FromArgb(0xFF, 0xFF, 0xFF)
        };

        for (int i - 0; i < acolor.Length; i++)
        {
             MenuItemColor mic = new MenuItemColor();
             mic.OwnerDraw     = true;
             mic.Color         = acolor[i];
             mic.Click        += ehGnClick;
             mic.fleasu reItem  += ehOnMeasureItem;
             mic.DrawItem      += ehOnDrawItem;
             mic.Break          = i % 4 == 0;

             menu.MenuItems.Add(mic);
        }
             // И наконец, создаем саму кнопку.

        ToolBarButton tbarbtn = new ToolBarButtonC);
        tbarbtn.ImageIndex = 4;
        tbarbtn.Style = ToolBarButtonStyle.DropDownButton;
        tbarbtn.DropDownMenu = menu;
        tbarbtn.ToolTipText = "Color";

        tbar.Buttons.Add(tbarbtn);
    }
    void MenuColorOnClick(object obj, EventArgs ea)
    {
             // Устанавливаем новый цвет текста.

        MenuItemColor mic = (MenuItemColor) obj;
        clrText = mic.Color;
        panel. Invalidate();

             // Изменяем изображение на кнопке,

        tbar.ImageList.Images[4] = CreateBitmapButton(clrText);
        tbar.Invalidate();
    }
```

см. след. стр.

```
    void MenuColorOnMeasureItem(object obj, MeasureItemEventArgs miea)
    {
        miea.ItemHeight = 18;
        miea.ItemWidth = 18;
    }
    void MenuColorOnDrawItem(object obj, DrawItemEventArgs diea)
    {
        MenuItemColor mic = (MenuItemColor) obj;
        Brush brush = new SolidBrush(mic.Color);

        Rectangle rect = diea.Bounds;

        rect.X += 1;
        rect.Y += 1;
        rect.Width -= 2;
        rect.Height -= 2;

        diea.Graphics.FillRectangle(brush, rect);
    }
    Bitmap CreateBitmapButton(Color clr)
    {
        Bitmap    bm    = new Bitmap(16, 16);
        Graphics grfx   = Graphics.FromImage(bm);
        Font     font   = new Font("Arial", 10, FontStyle.Bold);
        SizeF    sizef  = grfx.MeasureString("A", font);
        float    fScale = Math.Min(bm.Width / sizef.Width,
                                   bm.Height / sizef.Height);

        font = new Font(font.Name, fScale * font.SizeInPoints, font.Style);
        StringFormat strfmt = new StringFormat();
        strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;

        grfx.Clear(Color.White);
        grfx.DrawString("A", font, new SolidBrush(clr),
                        bm.Width / 2, bm.Height / 2, strfmt);
        grfx.Dispose();

        return bm;
    }
}
class MenuItemColor: MenuItem
{
    Color clr;

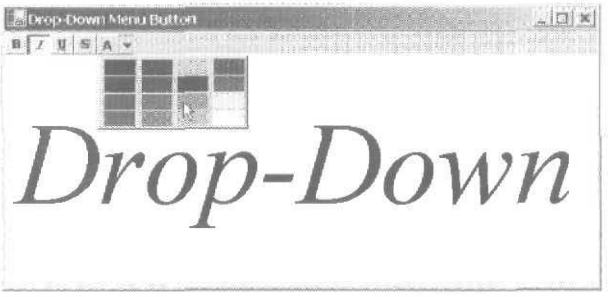
    public Color Color
    {
        get { return clr; }
        set { clr = value; }
    }
}
```

В этом файле также содержится переопределение класса *MenuItem*, добавляющее к нему свойство *Color*.

Расположенный в нижней части листинга метод *CreateBitmapButton* создает картинку размером 16x16 пикселов. Она состоит из буквы «А», выполненной шрифтом Arial. Цвет буквы устанавливается согласно переданному методу аргументу. Конструктор программы начинается с формирования начального изображения и добавления его в объект *ImageList*, созданный в программе ToggleButtons.

Затем создается контекстное меню, основанное на классе *MenuItemColor* который определяется в конце программы. Устанавливаются обработчики событий *Click*, *MeasureItem* и *DrawItem*. При создании кнопки ее свойству *Style* задается значение *ToolBarButtonStyle.DropDownButton*Свойству *DropDownMenu* сопоставляется созданное меню.

Обработчик события *MeasureItem* устанавливает размер элемента меню равным 18X18 пикселов, а обработчик события *DrawItem* чертит внутри этой зоны прямоугольник, оставляя вокруг него границу в 1 пиксел. Вот как выглядит получившееся меню:



Изображения в меню неквадратные, так как обработчик события *DrawItem* не учитывает дополнительное место, отведенное для флажка.

Обработчик события *MenuColorOnClick* устанавливает значение поля *clrText*, управляющего цветом текста в методе *PanelOnPaint*, и делает панель недействительной. В завершение создается новое изображение для кнопки.

И последнее: поскольку программа DropDownMenuButton унаследована от ToggleButtons, а ToggleButtons в качестве ресурса загружает растровое изображение, то в программе DropDownMenuButton нужно изменить пространство имен ресурса на «ToggleButtons», иначе он не будет загружен. (Эта проблема обсуждалась в главе 11.)

Стандартная панель может содержать обычные кнопки, кнопки-переключатели и выпадающие меню. Если вам требуется что-то еще (скажем, поле со списком), надо поступить несколько иначе: создать элемент управления типа *Panel,* контейнером которого будет служить форма, а он в свою очередь будет служить контейнером одного или нескольких элементов типа *ToolBar*, *ComboBox* или любых других.



ГЛАВА

21

Печать

Печать в приложении Windows Forms доставляет сравнительно мало хлопот, но именно *сравнительно;* она покажется простой и легкой, только когда за плечами есть опыт борьбы с API принтера в Win32. Печатать всегда сложнее, чем выводить текст и графику на экран. Дополнительные трудности заключаются в разнообразии типов принтеров, относительно небольшой скорости печати, наличии *у* принтера множества дополнительных приспособлений (подающих и приемных лотков), разных размеров бумаги и других прозаических проблем (таких как застрявшая бумага).

Изучение печати в Windows Forms — дело непростое, отчасти из-за того, что за печать отвечают несколько взаимосвязанных классов, которые, по-видимому, ссылаются друг на друга. Так, у класса *PrinterSettings* есть свойство типа *PageSettings,* а у класса *PageSettings* — свойство типа *PrinterSettings,* и это только начало: познакомившись с пространством имен *System.Drawing.Printing* поближе, вы найдете его похожим на зал с зеркальными стенами. При изучении печати по большей части приходится заниматься отбором нужных классов.

Хотя уже в начале главы вы получите достаточно сведений, чтобы в дальнейшем разбираться с печатью без посторонней помощи, вам, вероятно, захочется задействовать преимущества библиотеки стандартных диалоговых окон (о ней — в конце главы), помогающей писать приложения и стандартизировать их вид. Пространство имен *System.Windows.Forms* содержит классы, способные выводить стандартные диалоговые окна печати, параметры страницы и предварительного просмотра документа.

Давайте начнем наше путешествие с самих принтеров.

Принтеры и их параметры

Windows разрешает пользователю устанавливать несколько принтеров. (Точнее, *драйверы* для нескольких принтеров. Чтобы установить принтер в Windows, во-

все не обязательно подключать его к компьютеру.) Диалоговое окно Printers (Принтеры), вызываемое командой Settings (Настройка) из меню Start (Пуск), позволяет вывести список установленных принтеров. В любой момент времени один из принтеров, перечисленных в этом списке, является *принтером по умолчанию*. Принтер по умолчанию можно выбрать в диалоговом окне Printers,

Для программы Windows Forms описанием принтера является объект типа *PrinterSettings*. Подобно большинству объектов, о которых пойдет речь в этой главе (кроме стандартных диалоговых окон), он определяется в пространстве имен *System.Drawing.Printing*. У класса *PrinterSettings* есть только конструктор по умолчанию, который создает объект *PrinterSettings* для принтера по умолчанию;

Конструктор *PrinterSettings*

```
PrinterSettings ()
```

Например, оператор:

```
PrinterSettings prnset = new PrinterSettings();
```

создает новый экземпляр объекта *PrinterSettings,* ссылающийся на принтер по умолчанию.

Следующие три свойства предоставляют некоторые базовые сведения о принтере:

Свойства *PrinterSettings* (выборочно)

Тип	Свойство	Доступ
string	*PrinterName*	Чтение/запись
bool	*IsValid*	Чтение
bool	*IsDefaultPrinter*	Чтение

PrinterName — это строка, обычно содержащая имя и модель принтера, такая же строка выводится в диалоговом окне Printers. Вот несколько примеров:

```
HP LaserJet 1100 (MS)
NEC Silentwriter LC890 v47.0
Hewlett-Packard HP-GL/2 Plotter
Fax
```

Устанавливая принтер, пользователь может изменить его имя, поэтому имя принтера, показанное объектом *PrinterSettings*, может отличаться от стандартного.

При создании нового объекта *PrinterSettings* свойствам *IsValid* и *IsDefaultPrinter* обычно присваивается значение *true*. Но если в системе не установлен ни один принтер, свойство *PrinterName* возвращает строку «<no default printer>», а *IsValid* равно *false*.

Заметьте: свойство *PrinterName* изменяемое. Это означает, что если в него занести строку, идентифицирующую установленный принтер, то в остальные свойства объекта *PrinterSettings* также будут занесены значения для этого принтера. Ясно, что строка, заносимая в это свойство, должна совпадать с именем одного из установленных принтеров. Хотя в противном случае исключения не возникает, свойству *IsValid* будет присвоено значение *false*.

Чтобы не сделать ошибки при изменении свойства *PrinterName,* сначала нужно получить список установленных принтеров. Для этого служит единственное статическое свойство объекта *PrinterSettings:*

Статическое свойство *PrinterSettings*

Тип	Свойство	Доступ
PrinterSettings.StringCollection	*InstalledPrinters*	Чтение

Класс *StringCollection* определяется в классе *PrinterSettings,* реально это всего лишь массив неизменяемых строк. Допустим, свойство *InstalledPrinters* использовано так:

```
PrinterSettings.StringCollection sc = PrinterSettings.InstalledPrinters();
```

После этого будут доступны два свойства объекта *sc:*

Свойства *PrinterSettings.StringCollection*

Тип	Свойство	Доступ
int	*Count*	Чтение
string	*[]*	Чтение

Значение *scCount* —это число установленпык принтеров (если установленных принтеров нет, оно равно 0. *sc[0]*— это имя первого принтера, *sc[1]*— второго и т. д.

Можно не сохранять значение *InstalledPrinters* в переменной, а обращаться к самому свойству. Так, значение:

```
PrinterSettings.InstalledPrinters.Count
```

— это число установленных принтеров, а;

```
PrinterSettings.InstalledPrinters[1]
```

- имя второго принтера. Следующий код помещает имена всех установленных принтеров в объект *ComboBox* с именем *combo:*

```
foreach (string str in PrinterSettings.InstalledPrinters)
    combo.Items.Add(str);
```

Можно изменить принтер, на который ссылается объект *PrinterSettings,* поместив в свойство *PtinterName* строку из набора *StringCollection.* Если *sc* — переменная для хранения *StringCollection,* это делается так:

```
prnset.PrinterName = sc[2];
```

Можно присвоить значение *PrinterName* напрямую, проиндексировав свойство *InstalledPrinters:*

```
prnset.PrinterName = PrinterSettings.InstalledPrinters[2];
```

Если не возникнет серьезных ошибок, свойству *IsValid* в результате должно быть присвоено *true,* а *IsDefaultPrinter— false* (даже если занести в *PrinterName* имя принтера по умолчанию).

Подчеркну: если занести в *PrinterName* имя установленного принтера, в остальные свойства объекта *PrinterSettings* будут занесены значения для этого принтера,

Вот пара свойств, описывающих фундаментальные возможности принтера:

Свойства *PrinterSettings* (выборочно)

Тип	Свойство	Доступ
bool	*IsPlotter*	Чтение
bool	*SupportsColor*	Чтение
int	*LandscapeAngle*	Чтение

Если свойство *IsPlotter* равно *true*, по-видимому, растровые изображения на таком принтере лучше не печатать. Если принтер не поддерживает цветную печать, возможно, при выводе некоторых видов графики придется задействовать альтернативные цвета (например, для печати цветных диаграмм и карт лучше применить узорные кисти — см. о них главу 17). Свойство *LandscapeAngle* обычно равно 90° или 270°, но если принтер не поддерживает печать страниц в альбомной ориентации, оно будет равно 0.

Объект *PrinterSettings* ничего не говорит о технологии печати принтера (лазерная она, струйная или иная).

Три свойства объекта *PrinterSettings* возвращают наборы элементов и показывают доступные источники бумаги (лотки), перечень поддерживаемых принтером размеров бумаги (включая размеры конвертов) и список доступных разрешений:

Свойства *PrinterSettings* (выборочно)

Тип	Свойство	Доступ	Элементы
PrinterSettings.PaperSourceCollection	*PaperSources*	Чтение	*PaperSource*
PrinterSettings.PaperSizeCollection	*PaperSizes*	Чтение	*PaperSize*
PrinterSettings.PrinterResolutionCollection	*PrinterResolutions*	Чтение	*PrinterResolution*

Принцип работы этих свойств довольно прост. Все три класса (столбец Тип) определяются в классе *PrinterSettings*. У каждого есть по паре неизменяемых свойств: *Count*, соответствующее числу элементов набора, и индексатор, возвращающий объект, тип которого указан в столбце Элементы.

Например, свойство *PaperSources* по существу является набором объектов *PaperSource*. Значение:

```
prnset.PaperSources.Count
```

указывает число этих объектов в наборе. Индексируя свойство, можно ссылаться на любой из них. Таким образом:

```
prnset.PaperSources[2]
```

это третий объект типа *PaperSource* из набора *PrinterSettings.PaperSourceCollection*. Если в наборе меньше 3 элементов, то ссылка на третий элемент вызывает исключение.

Рассмотрим классы *PaperSource*, *PaperSize* и *PrinterResolution*. У класса *PaperSource* всего два свойства, оба неизменяемые:

Свойства *PaperSource*

Тип	Свойство	Доступ
string	*SourceName*	Чтение
PaperSourceKind	*Kind*	Чтение

Свойство *SourceName* содержит текст описания, имеющего значение для пользователя (например, «ручная подача бумаги»), а *PaperSourceKind* — это перечисление:

Перечисление *PaperSourceKind*

Член	Значение	Член	Значение
Upper	1	*TractorFeed*	8
Lower	2	*SmallFormat*	9
Middle	3	*LargeFormat*	10
Manual	4	*LargeCapacity*	11
Envelope	5	*Cassette*	14
ManualFeed	6	*FormSource*	15
AutomaticFeed	7	*Custom*	257

Свойство *PaperSources* объекта *PrinterSettings* — это *полный* набор источников бумаги принтера. Его значение *не* отражает источник бумаги, выбранный в данный момент как источник бумаги по умолчанию (об этом чуть ниже).

Свойство *PaperSizes* объекта *PrinterSettings* — это полный набор размеров бумаги, поддерживаемых принтером. Каждый элемент этого набора — объект типа *PaperSize* с четырьмя свойствами;

Свойства *PaperSize*

Тип	Свойство	Доступ
string	*PaperName*	Чтение/запись
int	*Width*	Чтение/запись
int	*Height*	Чтение/запись
PaperKind	*Kind*	Чтение

PaperName — это текстовая строка, имеющая значение для пользователя, например, «конверт номер 10». Свойства *Width* и *Height* указывают размер бумаги (конверта) в сотых долях дюйма. *PaperKind* — это перечисление, слишком большое, чтобы привести его здесь полностью (при последнем подсчете в нем было 117 членов). Для примера приведем несколько значений из этого перечисления, встречающихся в США и Европе:

Перечисление *PaperKind* (выборочно)

Член	Значение	Описание
Letter	1	8,5X11 дюймов
Legal	5	8,5X14 дюймов
Executive	7	7,25X10,5 дюймов
A4	9	210X297 мм
45	11	148X210 мм

Свойство *PrinterResolutions* объекта *PrinterSettings* — это набор объектов *PrinterResolution*. У класса *PrinterResolution* три свойства:

Свойства *PrinterResolution*

Тип	Свойство	Доступ
int	*X*	Чтение
int	*Y*	Чтение
PrinterResolutionKind	*Kind*	Чтение

PrinterResolutionKind — это очередное перечисление:

Перечисление *PrinterResolutionKind*

Член	Значение
Custom	0
Draft	-1
Low	-2
Medium	-3
High	-4

У каждого принтера набор *PrinterResolutions* состоит минимум из пяти элементов. Для четырех из них значения свойства *PrinterResolutionKind* равны соответственно *Draft*, *Low*, *Medium* и *High*, а значения свойств *X* и *Y* равны -1. Эти элементы не сопоставлены уникальным значениям разрешения принтера. Так, если принтер поддерживает только одно разрешение, этим четырем элементам будет соответствовать одно и то же разрешение.

Остальные элементы набора *PrinterResolutions* показывают реальные значения разрешения устройства, доступные на принтере. Для этих элементов значение *PrinterResolutionKind* равно *Custom*, а свойства *X* и *Y* указывают реальное разрешение в точках на дюйм.

Допустим, принтер поддерживает два разрешения: 600x600 и 1200x1200. В этом случае набор *PrinterResolutions* состоит из шести элементов, для двух из которых значение *PrinterResolutionKind* будет равно *Custom*. У одного элемента этого набора значения *X* и *Y* будут равны 600, у других значения *X* и *Y* будут равны 1200. Остальные четыре элемента — это *Draft*, *Low*, *Medium* и *High*, у них свойства *X* и *Y* будут равны -1.

Следующие свойства объекта *PrinterSettings* определяют печать многостраничных документов:

Свойства *PrinterSettings* (выборочно)

Тип	Свойство	Доступ
bool	*CanDuplex*	Чтение
Duplex	*Duplex*	Чтение/запись
int	*MaximumCopies*	Чтение
short	*Copies*	Чтение/запись
bool	*Collate*	Чтение/запись

Свойство *CanDuplex* равно *true,* если принтер может печатать на обеих сторонах листа одновременно. Если оно равно *true,* свойству *Duplex* может быть присвоено одно из этих значений:

Перечисление *Duplex*

Член	Значение
Simplex	1
Vertical	2
Horizontal	3
Default	-1

Значение *Simplex* задает одностороннюю печать, *Vertical* и *Horizontal* обозначают разные способы печати двухсторонних документов. *Vertical* указывает, что страницы должны складываться вертикально, как в обычной книге, а *Horizontal* задает печать страниц, которые складываются по горизонтали.

По умолчанию значение свойства *Copies* равно 1. Чтобы заставить драйвер принтера напечатать несколько копий документа, можно присвоить ему большее значение, но не больше *MaximumCopies.* Свойство *Collate* указывает порядок печати страниц каждой копии. Если при печати двух копий трехстраничного документа оно равно *false,* страницы будут напечатаны в порядке 1, 1, 2, 2, 3, 3, если же *true* — порядок печати будет 1, 2, 3, 1. 2. 3. Значение *Collate* по умолчанию зависит от принтера.

Если программно установить значения следующих свойств, ничего не произойдет. Они предназначены для применения с классом *PrintDialog,* который мы обсудим ниже:

Свойства *PrinterSettings* (выборочно)

Тип	Свойство	Доступ
PrintRange	*PrintRange*	Чтение/запись
int	*MinimumPage*	Чтение/запись
int	*MaximumPage*	Чтение/запись
int	*FromPage*	Чтение/запись
int	*ToPage*	Чтение/запись
bool	*PrintToFile*	Чтение/запись

Последнее свойство класса *PrinterSettings* — это объект типа *PageSettings,* который представляет еще один важный класс, определяемый в пространстве имен *System.Drawing.Printing:*

Свойства *PrinterSettings* (выборочно)

Тип	Свойство	Доступ
PageSettings	*DefaultPageSettings*	Чтение

Чуть ниже мы обсудим класс *PageSettings* подробнее. Он описывает характеристики печатаемой страницы. Так, у объекта *PrinterSettings* есть свойство *PaperSources,* содержащее набор всех доступных источников бумаги принтера. Свойство *PaperSource* объекта *PageSettings* указывает источник бумаги для некоторой страницы.

Свойство *DefaultPageSettings* объекта *PrinterSettings* определяет параметры страницы по умолчанию. Как я покажу ниже, при печати можно менять параметры страницы как для всего документа, так и для любой из его страниц по отдельности.

Методы объекта *PrinterSettings* позволяют взаимодействовать с кодом Win32, в частности, копировать данные из объекта *PrinterSettings* в структуры Win32 *DEVMODE* или *DEVNAMES* и обратно.

Другой метод объекта *PrinterSettings* интересен, даже если программа Windows Forms не взаимодействует с кодом Win32:

Методы *PrinterSettings* (выборочно)

```
Graphics CreateMeasurementGraphics()
```

Он возвращает то, что в Win32 называется *контекстом устройства* (informal ion device context). Объект *Graphics*, полученный при помощи *CreateMeasurementGraphics*, позволяет не только рисовать на странице принтера, но и получать сведения о нем. Этот метод в любое время (например, во время исполнения конструктора программы) выдаст дополнительные сведения о любом из установленных принтеров. В Windows Forms подобная информация имеет куда как меньшее значение, в основном из-за того, что в Windows Forms обработка шрифтов зависит от устройства в меньшей степени, чем в программах, использующих Win32API.

А теперь перейдем от *PrinterSettings* к *PageSettings*. Изучив этот класс, мы покорим второй базовый класс из пространства имен *System.Drawing.Printing*.

Параметры страницы

Класс *PageSettings* описывает параметры принтера, которые могут меняться от страницы к странице, поэтому есть соблазн рассматривать объект *PageSettings* в изоляции от других объектов. Однако любой объект *PageSettings* всегда связан с принтером. Если подумать, станет понятно, почему это так если объект *PageSettings* указывает, что страница должна быть напечатана на бумаге формата ledger (17×11 дюймов), лучше, чтобы принтер поддерживал именно этот формат.

Обычно программы обращаются к готовым объектам *PageSettings* (например, к свойству *DefaultPageSettings* объекта *PrinterSettings*, которое содержит такой объект), но конструктор этого класса также позволяет создавать новые объекты *PageSettings*:

Конструкторы *PageSettings*

```
PageSettings()
PageSettings(PrinterSettings prnset)
```

Первый конструктор создает объект *PageSettings* для принтера по умолчанию, а второй — для любого из установленных принтеров, заданного аргументом *PrinterSettings*. В обоих случаях созданный объект *PageSettings* содержит параметры страницы по умолчанию для заданного принтера.

Пользователи определяют параметры страницы по умолчанию для установленных принтеров в диалоговом окне со свойствами данного принтера (Printing Preferences), которое можно вызвать из диалогового окна Printers. Windows Forms может изменить эти параметры во время печати документа, но никакие изменения, внесенные Windows Forms, *не повлияют* на другие приложения. Допустим, пользователь установил альбомную ориентацию страницы в диалоговом окне Printing Preferences. Если после этого программа Windows Forms напечатает документ с книжной ориентацией страницы, действия программы не изменят альбомную ориентацию страницы, заданную в диалоговом окне Printing Preferences.

У класса *PageSettings* восемь свойств, семь из которых изменяемы:

Свойства *PageSettings*

Тип	Свойство	Доступ
PrinterSettings	*PrinterSettings*	Чтение/запись
bool	*Landscape*	Чтение/запись
Rectangle	*Bounds*	Чтение
Margins	*Margins*	Чтение/запись
bool	*Color*	Чтение/запись
PaperSource	*PaperSource*	Чтение/запись
PaperSize	*PaperSize*	Чтение/запись
PrinterResolution	*PrinterResolution*	Чтение/запись

Первое свойство *(PrinterSettings)* задает принтер, с которым ассоциированы эти параметры страницы. Если получить объект *PrinterSettings* из свойства *DefaultPageSettings* объекта *DefaultPageSettings* у объекта *PageSettings* в свойстве с именем *PrinterSettings* окажется объект, тождественный исходному *PrinterSettings*.

Иначе говоря, если создать объект *PrinterSettings* с именем *prnset,* оператор:

```
(prnset == prnset.DefaultPageSettings.PrinterSettings)
```

вернет *true*. Помните: эти объекты на самом деле — ссылки, поэтому любые изменения свойств *prnset* будут отражены в *prnset.DefaultPageSettings.PrinterSettings*.

Если же создать объект *PageSettings с* именем *pageset,* оператор:

```
(pageset == pageset.PrinterSettings.DefaultPageSettings)
```

вернет *false,* даже если все свойства соответствующих объектов исходно будут одинаковыми. Объект типа *PageSettings* ссылается на параметры конкретной страницы, что удобно, когда нужно изменить параметры некоторой страницы, не меняя параметров страницы по умолчанию, заданных для всего документа.

Остальные свойства *PageSettings* обычно используются просто для получения нужных сведений. Однако программа может изменять значения этих свойств (в определенных переделах), чтобы изменить способ печати страницы.

Так, свойство *Landscape* позволяет получить сведения об ориентации страницы: *false* указывает, что задана книжная ориентация страницы, а *true* — что альбомная. Если приложение должно печатать разные документы в зависимости от ориентации страницы, эти сведения позволяют выбрать, что печатать. Программа также может самостоятельно изменять это свойство без вмешательства пользователя.

Неизменяемое свойство *Bounds* — это объект *Rectangle,* который указывает размер страницы с учетом размера бумаги и значения свойства *Landscape,* выраженный в сотых долях дюйма. Так, для листа формата letter при книжной ориентации страницы значение *Bounds* составит (О, О, 850, 1100), для страницы аналогичного формата в альбомной ориентации — (0. О, 1100, 850).

Свойство *Margins* указывает ширину полей страницы по умолчанию. Исходно для всех полей страницы установлена ширина в 1 дюйм. Новый объект *Margins* создается с помощью конструкторов:

Конструкторы *Margins*

```
Margins()
Margins(int Left, int Right, int Top, int Bottom)
```

Четыре свойства этого класса указывают ширину полей в сотых долях дюйма:

Свойства *Margins*

Тип	Свойство	Доступ
int	*Left*	Чтение/запись
int	*Right*	Чтение/запись
int	*Top*	Чтение/запись
int	*Bottom*	Чтение/запись

Иногда пользователь запрещает печатать страницу в цвете, хотя принтер поддерживает цветную печать (скажем, из-за того, что кончился цветной картридж). Свойство *Color* объекта *PageSettings* указывает, хочет ли пользователь печатать страницу в цвете.

Три свойства объекта *PageSettings: PaperSource, PaperSize* и *PrinterResolution* — соответствуют трем свойствам класса *PrinterSettings: PaperSources, PaperSizes* и *PrinterResolutions, с* которыми вы уже знакомы (обратите внимание на форму множественного числа всех имен свойств класса *PrinterSettings).* Так, свойство *PaperSource* объекта *PageSettings* — это один из элементов набора *PaperSources,* составляющего одноименное свойство объекта *PrinterSettings.*

Если программа изменяет одно из этих свойств, убедитесь, что в изменяемое свойство помещается элемент из соответствующего набора. Пусть имеется объект типа *PageSettings — pageset.* Тогда код, изменяющий разрешение принтера на *Draft,* будет выглядеть как-то так:

```
foreach (PrinterResolution prnres in pageset.PrinterSettings.PrinterResolutions)
{
    if (prnres.Kind == PrinterResolutionKind.Draft)
        pageset.PrinterResolution = prnres;
}
```

Цикл *foreach* обрабатывает все элементы набора *PrinterResolutions* из объекта *PrinterSettings,* связанного с объектом *PageSettings.* Обнаружив совпадение, программа устанавливает свойство *PrinterResolution.* В свойство *PrinterResolution* объекта *PageSettings* должен быть помещен готовый объект *PrinterResolution,* поскольку у класса *PrinterResolution* нет открытого конструктора.

Программе иногда может потребоваться изменить значения свойств *PaperSource* или *PaperSize*. Скажем, вы разрабатываете программное средство для создания стандартных писем, которое вместе с письмами печатает конверты для них в одном задании печати. В этом случае потребуется менять значения свойств *PaperSource* и *PaperSize* в зависимости от параметров, заданных пользователем в приложении.

Значение свойства *Landscape* не влияет на свойство *PaperSize*. Если *Landscape* равно *false*, то значения *Width* и *Height* у *Bounds* равны соответствующим значениям свойства *PaperSize,* иначе значения свойств *Width* и *Height* взаимозаменяются, тогда как с аналогичными свойствами *PaperSize* этого не происходит

Пока мы так и не добрались до настоящей печати: для этого прежде нужно определить объект типа *PrintDocument*.

Определение документа для печати

Задание печати состоит из одной или нескольких страниц, печатаемых на принтере. Оно представлено объектом *PrintDocument*, у которого есть лишь конструктор по умолчанию:

Конструктор *PrintDocument*

```
PrintDocument()
```

В общем случае программа начинает процесс печати с создания объекта типа *PrintDocument:*

```
PrintDocument prndoc = new PrintDocument();
```

Для каждого задания печати можно создать новый объект *PrintDocument*. Если же пользователь выбирает принтер и устанавливает его параметры в стандартном диалоговом окне печати (или при помощи других средств), лучше сохранять заданные параметры в объекте *PrintDocument* и применять этот экземпляр объекта в течение всего сеанса работы программы. В этом случае объект *prndoc* определяется как поле и создается однократно.

У *PrintDocument* всего четыре свойства, но два содержат объекты типа *PrinterSettings* и *PageSettings*, так что в этом объекте заключено намного больше информации, чем может показаться:

Свойства *PrintDocument*

Тип	Свойство	Доступ
PrinterSettings	*PrinterSettings*	Чтение/запись
PageSettings	*DefaultPageSettings*	Чтение/запись
string	*DocumentName*	Чтение/запись
PrintController	*PrintController*	Чтение/запись

При создании нового объекта *PrintDocument* свойство *PrinterSettings* указывает принтер по умолчанию. При желании можно изменять свойство *PrinterSettings* или его отдельные свойства, скажем, так:

```
prndoc.PrinterSettings.Copies = 2;
```

Первоначально свойство *DefaultPageSettings* устанавливают на основе значения свойства *DefaultPageSettings* объекта *PrinterSettings*. Его тоже можно изменять, как и значения отдельных свойств этого свойства. Вот пример:

```
prndoc.DefaultPageSettings.Landscape = true;
```

Для нового объекта *PrintDocument* выражение:

```
prndoc.PrinterSettings == prndoc.DefaultPageSettings.PrinterSettings
```

возвращает *true,* но:

```
prndoc.DefaultPageSettings == prndoc.PrinterSettings.DefaultPageSettings
```

возвращает *false,* так как возможна ситуация, когда нужно изменить значение *DefaultPageSettings* для отдельного документа, не меняя параметров страницы по умолчанию для принтера,

Свойство *DocumentName* инициализируется текстовой строкой «document», которую скорее всего потребуется изменить. Это отображаемое имя задания печати, которое всегда используется для его идентификации (например, в окне, которое выводит список незавершенных заданий печати при печати соответствующих документов). Свойство *PrintController* мы обсудим чуть ниже,

У класса *PrintDocument* четыре открытых события:

События *PrintDocument*

Событие	Метод	Делегат	Аргумент
BeginPrint	*OnBeginPrint*	*PrintEventHandler*	*PrintEventArgs*
QueryPageSettings	*OnQueryPageSettings*	*QueryPageSettings EventHandler*	*QueryPageSettings EventArgs*
PnntPage	*OnPrintPage*	*PrintPageEventHandler*	*PrintPageEventArgs*
EndPrint	*OnEndPrint*	*PrintEventHandler*	*PrintEventArgs*

События *BeginPrint* и *EndPrint* происходят однократно для каждого задания печати, а *QueryPageSettings* и *PnntPage* — для каждой страницы задания. Обработчик события *PnntPage* указывает, остались ли еще страницы, которые должны быть напечатаны.

Как минимум, вам придется установить обработчик для события *PnntPage.* Если для каждой страницы нужно задавать собственные параметры (скажем, чтобы по очереди печатать письма и конверты в одном задании печати), придется также установить обработчик для события *QueryPageSettings.* Чтобы выполнить длинные процедуры инициализации или освобождения ресурсов, установите обработчики для событий *BeginPrint* и *EndPrint.* Аргументы всех этих обработчиков мы рассмотрим ниже.

Наконец, чтобы инициировать печать, нужно вызвать следующий метод (это единственный метод объекта *PrintDocument,* не связанный ни с одним событием):

Метод *PrintDocument*

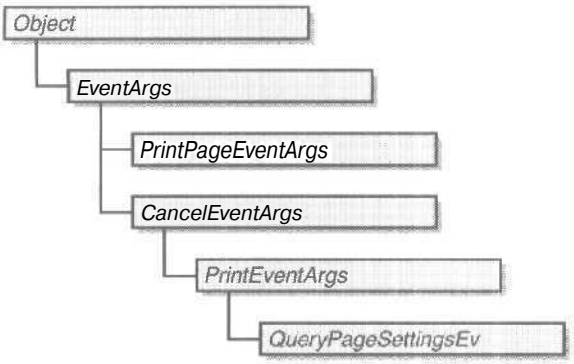
```
void Print<)
```

Метод *Print* не возвращает управление, пока программа не закончит печать документа. В это время приложение не реагирует на действия пользователя. Во время

исполнения метода *Print* вызываются обработчики события *PrintDocument*, подключенные этой программой. Сначала вызывается обработчик *BeginPrint*, затем — *QueryPageSettings* и *PrintPage* для каждой страницы и, наконец, — *EndPrint*.

Обработка событий объекта *PrintDocument*

На следующей схеме иерархии классов показаны объекты—потомки *EventArgs*, участвующие в обработке события *PrintDocument*:



Объект *CancelEventArgs* определяется в пространстве имен *System.ComponentModel*. Объект *PrintEventArgs*, связанный с событиями *BeginPrint* и *EndPrint*, обладает единственным свойством, унаследованным от *CancelEventArgs*:

Свойство *PrintEventArgs*

Тип	Свойство	Доступ
bool	*Cancel*	Чтение/запись

Обработчик события *BeginPrint* может присвоить свойству *Cancel* значение *true*, чтобы прервать исполнение задания печати (скажем, если задание требует больше памяти, чем доступно в системе).

Класс *QueryPageSettingsEventArgs* добавляет к *Cancel* еще одно свойство:

Свойство *QueryPageSettingsEventArgs*

Тип	Свойство	Доступ
PageSettings	*PageSettings*	Чтение/запись

Обработчик события *QueryPageSettings* может изменять свойства объекта *PageSettings*, чтобы подготовиться к соответствующему событию *PrintPage*.

У класса *PrintPageEventArgs* четыре неизменяемых свойства и два изменяемых:

Свойства *PrintPageEventArgs*

Тип	Свойство	Доступ
Graphics	*Graphics*	Чтение
bool	*HasMorePages*	Чтение/запись
bool	*Cancel*	Чтение/запись
PageSettings	*PageSettings*	Чтение

Свойства *PrintPageEventArgs* *(продолжение)*

Тип	Свойство	Доступ
Rectangle	*PageBounds*	Чтение
Rectangle	*MarginBounds*	Чтение

Для печати каждой страницы объект *Graphics* создается заново. Устанавливая свойства объекта *Graphics* (например, *PageUnit* или *PageScale*) для одной страницы, не ждите, что они будут действовать на следующие страницы. По умолчанию свойство *PageUnit* равно *GraphicsUnit.Display*, поэтому принтер кажется устройством с разрешением 100 dpi. Свойства *DpiX* и *DpiY* объекта *Graphics* отражают значение свойства *PrinterResolution* объекта *PageSettings*.

На входе в обработчик события печати страницы свойству *HasMorePages* всегда присваивается *false*. Если печатается многостраничный документ, по завершении обработчика события нужно присвоить этому свойству *true*, чтобы обработчик был вызван снова для печати следующей страницы. По завершении печати последней страницы следует оставить это свойство равным *false*.

Свойству *Cancel* обычно задается *false*. Если программе нужно прервать исполнение задания печати, этому свойству надо присвоить *true*. Задать свойству *Cancel* значение *true* и оставить *true* свойству *HasMorePages* — далеко не одно и то же: в первом случае ОС попытается остановить печать страниц, уже поставленных в очередь печати.

Свойство *PageSettings* служит для получения сведений о странице во время печати. Оно отражает все изменения параметров страницы, которые делаются при исполнении обработчика события *QueryPageSettings*.

Для удобства программиста в объект *PrintPageEventArgs* включен прямоугольник *PageBounds*, тождественный значению свойства *Bounds* объекта *PageSettings*, и *MarginBounds*, представляющий размеры страницы за вычетом полей, ширина которых задана свойством *Margins* объекта *PageSettings*. Ниже я расскажу об этих свойствах подробнее.

Следующая программа выводит простое диалоговое окно, позволяющее пользователю выбрать принтер из списка установленных принтеров.

PrinterSelectionDialog.cs

```
//-------------------------------------------------
// PrinterSelectionDialog.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class PrinterSelectionDialog: Form
{
     ComboBox combo;

     public PrinterSelectionDialog()
```

см. след. стр.

```
{
    Text = "Select Printer";

    FormBorderStyle = FormBorderStyle.FixedDialog;
    ControlBox      = false;
    MaximizeBox     * false;
    MinimizeBox     * false;
    ShowInTaskbar   = false;
    StartPosition   = FormStartPosition.Manual;
    Location        = ActiveForm.Location +
                          SystemInformation.CaptionButtonSize +
                          SystemInformation.FrameBorderSize;

    Label label    = new Label();
    label.Parent   = this;
    label.Text     = "Printer:";
    label.Location = new Point(8, 8);
    label.Size     = new Size(40, 8);

    combo = new ComboBox();
    combo.Parent   = this;
    combo.DropDownStyle = ComboBoxStyle.DropDownList;
    combo.Location * new Point(48, 8);
    combo.Size     = new Size(144, 8);

        // Добавить установленные принтеры в раскрывающийся список.

    foreach (string str in PrinterSettings.InstalledPrinters)
        combo.Items.Add(str);

    Button btn   = new Button();
    btn.Parent   = this;
    btn.Text     - "OK";
    btn.Location = new Point(40, 32);
    btn.Size     = new Size(40, 16);
    btn.DialogResult = DialogResult.OK;
    AcceptButton = btn;

    btn  - new Button();
    btn.Parent = this;
    btn.Text = "Cancel";
    btn. Location - new Point(120, 32);
    btn.Size    = new Size(40, 16);
    btn.DialogResult = DialogResult.Cancel;

    CancelButton = btn;

    ClientSize = new Size(200, 56);
    AutoScaleBaseSize = new SIze(4, a);
}
```

см. след. стр.

```
    public string PrinterName
    {
        set { combo.SelectedItem = value; }
        get { return (string) combo.SelectedItem; }
    }
}
```

Список относится к типу *DropDownList*, поэтому пользователю запрещено вводить что-либо в поле редактирования. Изменяемое свойство *PrinterName* позволяет программе инициализировать выбранный элемент списка именем принтера по умолчанию и получить выбранный пользователем элемент.

Такое же диалоговое окно вы видите и в программе PrintThreePages. Чтобы пользователь смог инициировать печать, она создает меню с вложенным меню File и командой Print. Обработчик элемента меню Print выводит диалоговое окно, где пользователь может выбрать принтер. Программа подключает обработчики событий *QueryPageSettings* и *PrintPage*, чтобы напечатать три страницы. В центре каждой страницы крупным шрифтом написан ее номер. Смеха ради программа устанавливает разрешение «draft» для всего документа и попеременно назначает для страниц то книжную, то альбомную ориентацию.

PrintThreePages.cs

```
//--------------------------------------------
// PrintThreePages.cs © 2001 by Charles Petzold
//--------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class PrintThreePages: Form
{
    const int iNumberPages = 3;
    int       iPageNumber;

    public static void Main()
    {
        Application.Run(new PrintThreePages());
    }
    public PrintThreePagesO
    {
        Text = "Print Three Pages";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&File");
        Menu.MenuItems[0].MenuItems.Add("&Print...",
                                  new EventHandler(MenuFilePrintOnClick));
    }
    void MenuFilePrintOnClick(object obj, EventArgs ea)
```

см. след. стр.

```
{
        // Создать объект PrintDocument.

    PrintDocument prndoc = new PrintDocument();

        // Создать диалоговое окно и установить свойство PrinterName.

    PrinterSelectionDialog dlg = new PrinterSelectionDialog(); ;
    dlg.PrinterName * prndoc.PrinterSettings.PrinterName;

        // Показать диалоговое окно и выйти из программы,
        // если не нажата кнопка OK.

    if (dlg.ShowDialog() != DialogResult.OK)
        return;

        ft Занести в PrintDocument имя выбранного принтера.

    prndoc.PrinterSettings.PrinterName = dlg.PrinterName;

        // Установить для принтера разрешение "draft".

    foreach (PrinterResolution prnres in
                        prndoc.PrinterSettings.PrinterResolutions)
    {
        if (prnres.Kind == PrinterResolutionKind.Draft)
            prndoc.DefaultPageSettings.PrinterResolution = prnres;
    }

        //Установить остальные свойства объекта PrintDocument.

    prndoc.DocumentName = Text;
    prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
    prndoc.QueryPageSettings += new QueryPageSettingsEventHandler
                                    (OnQueryPageSettings);
        // Начать печать.
    iPageNumber = 1;
    prndoc.Print();
}
void OnQueryPageSettings(object obj, QueryPageSettingsEventArgs qpsea)
{
    if (qpsea.PageSettings.PrinterSettings.LandscapeAngle != 0)
        qpsea.PageSettings.Landscape ^= true;
}
void OnPrintPage(object obj, PrintPageEventArgs ppea)
{
    Graphics grfx  = ppea.Graphics;
    Font    font  = new Font("Times New Roman", 360);
    string   str   = iPageNumber.ToString();
    SizeF    sizef = grfx.MeasureString(str, font);
```

См. след. стр.

печати — 8 дюймов, а высота — 10,6. Суммарная ширина областей, недоступных для печати справа и слева, — 0,5 дюйма, а суммарная ширина аналогичных областей сверху и снизу — 0,4 дюйма.

Однако из этого вовсе не следует, что слева и справа на моем принтере недоступны для печати равные области шириной по 0,25 дюйма, а сверху и снизу — области шириной по 0,20 дюйма. В зависимости от принтера ширина областей, недоступных для печати справа и слева, а также сверху и снизу, может быть неодинаковой,

В этой ситуации нам нужен прямоугольник, описывающий область печати относительно полного размера страницы. Увы, для расчета такого прямоугольника у нас мало информации. В объектах *PrinterSettings*, *PageSettings*, *PrintDocument* и *PrintPageEventArgs* нет данных о распределении областей, недоступных для печати, по краям страницы.

Если вас устраивают приближенные значения, можно рассчитать размеры прямоугольника, описывающего площадь страницы с учетом заданных пользователем полей, относительно *VisibleClipBounds* (значит, рассчитанный таким образом прямоугольник может быть использован рисующими методами объекта *Graphics)*. Если объект *PrintPageEventArgs* назван *ppea*, а объект *Graphics* — *grfx,* выражение:

```
(ppea.PageBounds.Width - grfx.VisibleClipBounds.Width) / 2
```

даст приблизительную ширину области, недоступной для печати с левого края страницы, а:

```
(ppea.PageBounds.Height - grfx.VisibleClipBounds.Height) / 2
```

— аналогичное значение для правого края страницы. Вычитание этой пары значений соответственно из *ppea.MarginBounds.Left* и *ppea.MarginBounds.Top* позволяет получить координаты точки, приблизительно соответствующей верхнему левому углу области печати страницы, рассчитанной с учетом ширины пользовательских полей.

Вот расчет размеров прямоугольника с учетом ширины пользовательских полей:

```
RectangleF rectf = new RectangleF(
     ppea.MarginBounds.Left -
          (ppea.PageBounds.Width - grfx.VisibleClipBounds.Width) / 2,
     ppea.MarginBounds.Top -
          (ppea.PageBounds.Height - grfx.VisibleClipBounds.Height) / 2,
     ppea.MarginBounds.Width,  ppea.MarginBounds.Height);
```

Еще раз: это приблизительный расчет, так как он основан на допущении о равенстве ширины недоступных для печати полей слева и справа, а также сверху и снизу. но ничего лучшего при помощи интерфейса Windows Forms сделать нельзя,

Я использую эти расчеты в следующей программе, которая просто рисует прямоугольник и соединяет его углы отрезками.

PrintWithMargins.cs

```
//--- -- ------------------- -----------------
// PrintWithMargins.cs e 2001 by Charles Petzold
//-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class PrintWithMargins: Form
{
    public static void Main()
    {
        Application. Яш (new PrintWithMargins());
    }
    public PrintWithMargins()
    {
        Text * "Print with Margins";
        Menu = new MainMenu();
        Menu.MenuItems.Add("&File");
        Menu.MenuItems[0].MenuItems.Add("&Print...",
                                    new EventHandler(MenuFilePrintOnClick));
    }
    void MenuFilePrintOnClick(object obj, EventArgs ea)
    {
            // Создать объект PrintDocument.

        PrintDocument prndoc = new PrintDocument();

            // Создать диалоговое окно и определить свойство PrinterName.

        PrinterSelectionDialog dlg = new PrinterSelectionDialog();
        tflg. PrinterName = prndoc. PrinterSettings.PrinterName;

            // Показать диалоговое окно и выйти из программы,
            // если нажата кнопка, отличная от OK.

        if (dlg.ShowDialog() != DialogResult.OK)
            return;

            // Занести в PrintDocument имя выбранного принтера.

        prndoc.PrinterSettings.PrinterName = dlg.PrinterName;

            // Установить остальные свойства объекта PrintDocument
            // и перейти к печати.

        prndoc.DocumentName = Text;
        prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
```

см. след. стр.

```
        prndoc.Print();
    }
    void OnPrintPage(object obj, PrintPageEventArgs ppea)
    {
        Graphics  grfx  = ppea.Graphics;
        RectangleF rectf = new RectangleF(
            ppea.MarginBounds.Left -
            (ppea.PageBounds.Width - grfx.VisibleClipBounds.Width) / 2,
            ppea.MarginBounds.Top -
            (ppea.PageBounds.Height - grfx.VisibleClipBounds.Height) / 2,
            ppea.MarginBounds.Width,
            ppea.MarginBounds.Height);

        grfx.DrawRectangle(Pens.Black, reetf.X, rectf.Y,
                                       rectf.Width, rectf.Height);

        grfx.DrawLine(Pens.Black, rectf.Left, rectf.Top,
                                  rectf.Right, rectf.Bottom);

        grfx.DrawLine(Pens.Black, rectf.Right, rectf.Top,
                                  rectf.Left, rectf.Bottom);
    }
}
```

Если такая аппроксимация вас не устраивает, придется задействовать функцию Win32 *GetDeviceCaps* с аргументами *PHYSICALOFFSETX* и *PHYSICALOFFSETY*.

Если вам удобнее другие единицы измерения (не 1/100 дюйма), можно преобразовать значения *PageBounds* и *MarginBounds* при помощи класса *PrinterUnitConvert*. У него один статический метод — ***Convert,*** который может быть определен в шести версиях:

Статический метод *Convert* класса *PrinterUnitConvert*

```
int Convert(int iValue, PrinterUnit puFrom, PrinterUnit puTo)
double Convert(double dValue, PrinterUnit puFrom, PrinterUnit puTo)
Point Convert(Point pt, PrinterUnit puFrom, PrinterUnit puTo)
Size Convert(Size size, PrinterUnit puFrom, PrinterUnit puTo)
Rectangle Convert(Rectangle rect, PrinterUnit puFrom, PrinterUnit puTo)
Margins Convert(Margins margins, PrinterUnit puFrom, PrinterUnit puTo)
```

PrinterUnit — это перечисление:

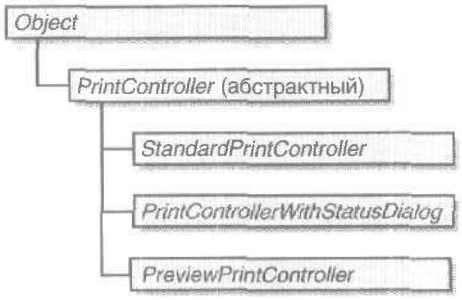
Перечисление *PrinterUnit*

Член	Значение
Display	0
ThousandthsOfAnInch	1
HundredthsOfAMillimeter	2
TenthsOfAMillimeter	3

Член *Display* указывает, что в качестве единиц измерения используются сотые доли дюйма,

Класс *PrintController*

Выше, говоря о классе *PrintDocument*, я намеренно пропустил свойство *PrintController*. По умолчанию в него помещают экземпляр объекта — потомка абстрактного класса *PrintController* («контроллер печати»). Вот иерархия классов, к которой принадлежат эти объекты:



Класс *PrintController* определяет четыре метода:

Методы *PrintController*

```
void OnStartPrint(PrintDocument prndoc, PrintEventArgs pea)
Graphics OnStartPage(PrintDocument prndoc, PrintPageEventArgs ppea)
void OnEndPage(PrintDocument prndoc, PrintPageEventArgs ppea)
void OnEndPrint(PrintDocument prndoc, PrintEventArgs pea)
```

Как вы уже знаете, когда программа инициирует печать, вызывая метод *Print* объекта *PrintDocument,* этот объект реагирует срабатыванием четырех событий, определенных в классе *PrintDocument: BeginPrint, QueryPageSettings, PrintPage* и *EndPrint*.

Но *PrintDocument* также вызывает четыре метода объекта *PrintController* из свойства *PrintController,* после срабатывания своего события *BeginPrint* — метод *OnStartPrint*, до и после срабатывания каждого события *PrintPage* — соответственно методы *OnStartPage* и *OnEndPage* и, наконец, после срабатывания своего события *OnEndPrint* — метод *OnEndPrint*.

Метод *OnStartPage* объекта *PrintController,* в частности, отвечает за получение объекта *Graphics*, который в конечном счете передается обработчику события *PrintPage* (обратите внимание на значение, возвращаемое методом *OnStartPage).* В сущности объект *Graphics* определяет место, куда метод *PrintPage* выводит графическую информацию,

Обычно выводимая графика поступает на принтер, за что и отвечает объект *PrintController.* В отличие от *PrintController,* у объекта *PreviewPrintController* другие функции: он создает объект *Graphics* на основе битовой карты, представляющей страницу принтера. Именно так реализован в Windows Forms предварительный просмотр печатаемого документа (об этом мы еще поговорим ближе к концу главы).

По умолчанию свойство *PrintController* объекта *PrintDocument* содержит объект типа *PrintControllerWithStatusDialog*. Имя этого объекта раскрывает еще одну функцию объекта «контроллер печати»; он отображает диалоговое окно с именем печатаемого документа и номером страницы, которая печатается в данный момент.

Если это диалоговое окно не нужно, поместите в свойство *PrintController* объекта *PrintDocument* объект типа *StandardPrintController*, который делает то же самое, что и *PrintControllerWithStatusDialog*, но при этом не выводит диалоговое окно.

Если нужно отображать ход печати не в диалоговом окне, а как-то иначе, можно породить соответствующий класс: от *StandardPrintController*. Вот пример программы для управления печатью, отображающей состояние печати на панели строки состояния.

StatusBarPrintController.cs

```
//-----------------------------------------------------
// StatusBarPrintController.cs © 2001 by Charles Petzold
//_____________________,_____________________________________
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class StatusBarPrintController: StandardPrintController
<
     StatusBarPanel  statpanel;
     int             iPageNumber;
     string          strSaveText;

     public StatusBarPrintController(StatusBarPanel sbp): base(>
     {
          statpanel = sbp;
      }
     public override void OnStartPrint(PrintDocument prndoc,
                                       PrintEventArgs pea)
     {
          strSaveText = statpanel.Text;       // Слово "Готов" или что-то
                                              // в этом роде
          statpanel.Text = "Starting printing";
          iPageNumber = 1;

          fease.OnStartPririt(prndoc, pea);
     }
     public override Graphics OnStartPage(PrintDocument prndoc,
                                          PrintPageEventArgs ppea)
     {
          statpanel.Text = "Printing page " + iPageNumber++;

          return base.OnStartPage(prndoc, ppea);
     }
     public override void OnEndPage(PrintDocument prndoc,
                                    PrintPageEventArgs ppea)
    Ч
          base.OnEndPage(prndoc, ppea);
     }
```

см. след. стр.

```
    public override void OnEndPrint(PrintDocument prndoc,
                                    PrintEventArgs pea)
    {
        statpanel.Text * strSaveText;

        base.OnEndPrint(prndoc, pea);
    }
}
```

Заметьте: этот класс не только переопределяет все четыре метода *StandardPrintController*, но и обеспечивает вызов соответствующих методов из базового класса. Это гарантирует, что объект «контроллер печати* не прекратит выполнять свои обычные действия. Единственное усовершенствование, реализованное в этой версии программы,— постоянное обновление панели строки состояния Объект «панель» нужен для конструктора класса.

Бот версия PrintThreePages, которая вместо диалогового окна для выбора принтера создает строку состояния с одной панелью,

PrintWithStatusBar.cs

```
//-------------------------------------------------
// PrintWithStatusBar.cs © 2001 by Charles Petzold
//-- -------- --------- -------- ------ ---------- -----
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class PrintWithStatusBar: Form
{
    StatusBar      sbar;
    StatusBarPanel sbarpanel;

    const int iNumberPages = 3;
    int       iPageNumber;
    public static void Main()
    {
        Application.Run(new PrintWithStatusBar());
    }
    public PrintWithStatusBar()
    {
        Text = "Print with Status Bar";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&File");
        Menu.MenuItems[0].MenuItems.Add("&Print",
                                        new EventHandler(MenuFilePrintOnClick));

        sbar * new StatusBar();
        sbar.Parent = this;
```

см. след. стр.

```
        sbar.ShowPanels = true;

        sbarpanel = new StatusBarPanel();
        sbarpanel.Text - "Ready";
        sbarpanel.Width = Width / 2;
        sbar.Panels.Add(sbarpanel);
    }
    void MenuFilePrintOnClick(object obj, EventArgs ea)
    {
        PrintDocument prndoc   = new PrintDocument();
        prndoc.DocumentName    = Text;
        prndoc.PrintController = new StatusBarPrintController(sbarpanel);
        prndoc.PrintPage      += new PrintPageEventHandler(OnPrintPage);

        iPageNumber = 1;
        prndoc.Print();
    }
    void OnPrintPage(object obj, PrintPageEventArgs ppea)
    {
        Graphics grfx  = ppea.Graphics;
        Font     font  = new Font("Times New Roman", 360);
        string   str   = iPageNumber.ToString();
        SizeF    sizef = grfx.MeasureString(str, font);

        grfx.DrawString(str, font, Brushes.Black,
                  (grfx.VisibleClipBounds.Width - sizef.Width) / 2,
                  (grfx.VisibleClipBounds.Height - sizef.Height) / 2);

        System.Threading.Thread.Sleep(1000);

        ppea.HasMorePages = iPageNumber < iNumberPages;
        iPageNumber += 1;
    }
}
```

Определяя свойство *PrintDocument* в ответ на выбор команды меню, эта версия программы также устанавливает свойство *PrintController*:

```
prndoc.PrintController = new StatusBarPrintController(statpanel);
```

Поскольку три страницы печатаются слишком быстро, чтобы успеть убедиться в правильной работе программы, я вставил в *OnPrintPage* вызов метода *Sleep* из класса *Thread*.

Единственная проблема при этом способе вывода сведений о состоянии принтера — обманчивость такой строки состояния. В отсутствие модального окна пользователю может показаться, что приложение должно реагировать на его действия. Как же! Приложение не может реагировать на действия пользователя, пока метод *Print* объекта *PrintDocument* не вернет управления. Если программа отображает статус принтера в строке состояния, следует реализовать печать в фоновом режиме, для чего нужен второй поток.

Стандартное диалоговое окно печати

Объект *PrintDialog* входит в набор стандартных диалоговых окон, определенных в пространстве имен *System.Windows.Forms*. Это окно позволяет пользователю выбирать принтер и изменять его параметры, а также задавать печать всего документа целиком, диапазона страниц или выделенной части документа (т. *с.* части, выделенной подсветкой).

Новый объект *PrintDialog* поволяет создать его конструктор по умолчанию:

Конструктор *PrintDialog*

```
PrintDialog()
```

Нужно инициализировать одно из следующих свойств (но не оба одновременно):

Свойства *PrintDialog* (выборочно)

Тип	Свосто	Доступ
PrintDocument	*Document*	Чтение/запись
PrinterSettings	*PrinterSettings*	Чтение/запись

Лучше установить свойство *Document,* после чего объект *PrintDialog* установит собственное свойство *PrinterSettings,* используя значение одноименного свойства *PrinterSettings* объекта *PrintDocument*.

Большая часть дополнительных возможностей объекта *PrintDialog* касается выбора страниц для печати. Пользователь может напечатать весь документ, диапазон страниц или выделенную в данный момент часть документа. Диалоговое окно *PrintDialog* позволяет сделать этот выбор при помощи переключателя, который можно установить в одно из положений: All, Pages и Selection, По умолчанию активизировано только AH, в которое и установлен переключатель.

Но можно активизировать и положения Pages и Selection. Это позволяеют сделать следующие свойства (кроме того, они позволяют активизировать ряд других функций диалогового окна печати):

Свойства *PrintDialog* (выборочно)

Тип	Свойство	Доступ	Значение по умолчанию
bool	*AllowSelection*	Чтение/запись	*False*
bool	*AllowSomePages*	Чтение/запись	*False*
bool	*AllowPrintToFile*	Чтение/запись	*True*
bool	*PrintToFile*	Чтение/запись	*False*
bool	*ShowNetwork*	Чтение/запись	*True*
bool	*ShowHelp*	Чтение/запись	*False*

Если свойству *ShowHelp* присваивается значение *true,* надо подключить и обработчик для события *HelpRequest* (унаследованного от класса *CommonDialog*). Свойство *AllowPrintToFile* активизирует флажок, позволяющий печатать в файл, а свойство *PrintToFile* показывает, установлен ли этот флажок.

Если активно положение Pages, пользователь может выбрать диапазон страниц, введя номера первой и последней страниц в поля From и To. Первоначальные

значения From и To можно задать соответственно как минимальное и максимальное значения этих полей, но не как свойства объекта *PrintDialog*, определяемые Б объекте *PrinterSettings*. После установки свойства *Document* объекта *PrintDialog*, свойство *PrinterSettings* объекта *PrintDialog* позволяет ссылаться на такие свойства:

Свойства *PrinterSettings* (выборочно)

Тип	Свосто	Доступ
PrintRange	*PrintRange*	Чтение/запись
int	*MinimumPage*	Чтение/запись
int	*MaximumPage*	Чтение/запись
int	*FromPage*	Чтение/запись
int	*ToPage*	Чтение/запись
bool	*PrintToFile*	Чтение/запись

Свойство *PrintRange* — это перечисление типа *PrintRange*:

Перечисление *PrintRange*

Член	Значение
AllPages	0
Selection	1
SomePages	2

Возможно, у вас возникнет желание задать *MinimumPage* значение 1, а *MaximumPage* — значение, равное числу страниц документа; аналогичными значениями можно инициализировать и свойства *FromPage* и *ToPage*. Но для некоторых приложений (типа клонов Notepad) это не лучшая идея: открыв диалоговое окно *PrintDialog*, пользователь может выбрать другой принтер, изменить ориентацию и размер страницы, а также другие параметры, любой из которых может повлиять на число страниц печатаемого документа.

Как и в случае любого стандартного диалогового окна, после инициализации объекта *PrintDialog* вызывают его метод *ShowDialog*, который возвращает одно из значений перечисления *DialogResult*. После возврата управления объектом *PrintDialog* свойство *PrintRange* содержит сведения о выборе, сделанном пользователем. Если он выбрал печать диапазона страниц, значения свойств *FromPage* и *ToPage* указывают границы диапазона.

Начнем с простого. Следующая программа — это еще одна версия PrintThreePages, использующая объект *PrintDialog*.

SimplePrintDialog.cs

```
//-----------------------------------------------
// SimplePrintDialog.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;
```

см. след. стр.

```
class PrintDialogHelloWorld: Form
{
     const int iNumberPages = 3;
     int       iPagesToPrint, iPageNumber;

     public static void Main()
     {
          Application.Run(new PrintDialogHelloWorld());
     }
     public PrintDialogHelloWorld()
     {
          Text = "Simple PrintDialog";

          Menu = new MainMenu();
          Menu.MenuItems.Add("&File");
          Menu.MenuItems[0].MenuItems.Add("&Print...",
                                  new EventHandler(MenuFilePrintOnClick));
     }
     void MenuFilePrintOnClick(object obj, EventArgs ea)
     {
               // Создать обьекты PrintDocument и PrintDialog.

          PrintDocument prndoc = new PrintDocument();
          PrintDialog   prndlg = new PrintDialog();
          prndlg.Document = prndoc;

               // Разрешить выбор диапазона страниц.

          prndlg.AllowSomePages = true;
          prndlg.PrinterSettings.MinimumPage = 1;
          prndlg.PrinterSettings.MaximumPage = iNumberPages;
          prndlg.PrinterSettings.FromPage = 1;
          prndlg.PrinterSettings.ToPage = iNumberPages;

               // Если диалоговое окно возвращает OK, перейти к печати.

          if(prndlg.ShowDialog() == DialogResult.OK)
          {
               prndoc.DocumentName = Text;
               prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);

                    // Определить набор страниц для печати.

               switch (prndlg.PrinterSettings.PrintRange)
               {
               case PrintRange.AllPages:
                    iPagesToPrint = iNumberPages;
                    iPageNumber = 1;
                    break;
```

см. след. стр.

```
            case PrintRange.SomePages:
                iPagesToPrint = 1 + prndlg.PrinterSettings.ToPage -
                                    prndlg.PrinterSettings.FromPage;
                iPageNumber = prndlg.PrinterSettings.FromPage;
                break;
            }
            prndoc.Print();
        }
    }
    void OnPrintPage(object obj, PrintPageEventArgs ppea)
    {
        Graphics grfx  = ppea.Graphics;
        Font     font  = new Font("Times New Roman", 360);
        string   str   = iPageNumber.ToString();
        SizeF    sizef = grfx.MeasureString(str, font);

        grfx.DrawString(str, font, Brushes.Black,
                (grfx.VisibleClipBounds.Width - sizef.Width) / 2,
                (grfx.VisibleClipBounds.Height - sizef.Height) / 2);

        iPageNumber += 1;
        iPagesToPrint -= 1;
        ppea.HasMorePages = iPagesToPrint > 0;
    }
}
```

Здесь сложнее всего реализовать возможность выбора диапазона страниц для печати. Значение поля *iNumberPages* жестко запрограммировано и равно 3. Но вместе с *iPageNumber* я определил новое поле — *iPagesToPrint,* способное принимать значения от 1 до *iNumberPages.*

Перед вызовом *PrintDialog* программа устанавливает все свойства этого объекта (в том числе и те, что устанавливаются у *PrinterSettings,* а не у самого *PrintDialog*), имеющие отношение к диапазону страниц. После возврата управления объектом *PrintDialog* программа проверяет значение *PrintRange* и инициализирует поля *iPagesToPrint* и *iPageNumber* (номер первой страницы для печати) соответствующими значениями.

Некоторые возможности, предоставляемые объектом *PrintDialog,* обрабатываются автоматически. Так, если установить флажок Print To File, откроется диалоговое окно, где нужно ввести имя файла, после чего графическая информация, предназначенная для принтера, будет сохранена в этом файле,

Немного поэкспериментировав, вы увидите, что при каждом вызове *PrintDialog* всем параметрам возвращаются значения по умолчанию. Вероятно, вам приходилось сталкиваться с подобным раздражающим поведением в некоторых приложениях для Windows. Если к компьютеру подключено несколько принтеров, случается, когда из какого-то приложения постоянно печатают па принтере, отличном от принтера по умолчанию. Если один раз выбрать этот принтер в *PrintDialog,* хотелось бы. чтобы диалоговое окно показывало его и при следующих вызовах. То же верно для других параметров, заданных в этом диалоговом окне,

Чтобы сделать выбранные значения постоянными, храните объекты *PrintDialog* и *PrintDocument* в виде полей. Для этого просто перенесите эти операторы за пределы метода *MenuFilePrintOnClick*:

```
PrintDocument prndoc = new PrintDocument();
PrintDialog   prndlg = new PrintDialog();
```

Затем переместите этот оператор в конструктор:

```
prndlg.Document = prndoc;
```

Я буду использовать этот прием в остальных программах из этой главы,

Настройка страницы

PageSetupDialog — второе стандартное диалоговое окно, имеющее отношение к печати. Обычно оно позволяет пользователю задавать ширину полей, ориентацию страницы, источники и размеры бумаги. Но оно также позволяет выбирать принтер по умолчанию и настраивать параметры принтера. У объекта *PageSetupDialog* только один конструктор:

Конструктор *PageSetupDialog*

```
PageSetupDialog()
```

Создав объект, можно установить одно (и только одно) из этих свойств:

Свойства *PageSetupDialog* (выборочно)

Тип	Свойство	Доступ
PrintDocument	*Document*	Чтение/запись
PrinterSettings	*PrinterSettings*	Чтение/запись
PageSettings	*PageSettings*	Чтение/запись

Рекомендуется устанавливать свойство *Document*, после чего *PageSetupDialog* поместит в свойства *PrinterSettings* и *PageSettings* соответствующие значения из заданного объекта *PrintDocument*. Чтобы все работало правильно, надо использовать один и тот же объект *PrintDocument* для *PageSetupDialog* и *PrintDialog*.

Вот остальные свойства объекта *PageSetupDialog*:

Свойства *PageSetupDialog* (выборочно)

Тип	Свойство	Доступ
bool	*AllowMargins*	Чтение/запись
bool	*AllowOrientation*	Чтение/запись
bool	*AllowPaper*	Чтение/запись
bool	*AllowPrinter*	Чтение/запись
bool	*ShowNetwork*	Чтение/запись
boot	*ShowHelp*	Чтение/запись
Margins	*MinMargins*	Чтение/запись

Все свойства типа *bool*, кроме *ShowHelp*, по умолчанию равны *true*. Если присвоить им *false*, некоторые возможности диалогового окна будут отключены. Кнопка Network находится в дополнительном диалоговом окне, которое открывается по щелчку кнопки Printer. По умолчанию все значения свойства *MinMargins* равны 0,

Изменения, вносимые пользователем в диалоговом окне *PageSetupDialog*, отражаются объектом *PageSettings*, который диалоговое окно получает из объекта *PrintDocument*.

ImagePrint наследует программе ImageIO из главы 16 (она в свою очередь происходит от ImageOpen). У этой программы в меню File добавлены команды Page Setup и Print. Теперь она может загружать, сохранять и печатать растровые изображения,

ImagePrint.cs

```
//---------------------------------------
// ImagePrint.cs e 2001 by Charles Petzold
//
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class ImagePrint: ImageIO
{
     PrintDocument   prndoc - new PrintDocument();
     PageSetupDialog setdlg = new PageSetupDialog();
     PrintDialog     prndlg = new PrintDialog();

     MenuItem miFileSet, miFilePrint, miFileProps;

     public new static void Main()
     {
          Application.Run(new ImagePrint());
     )
     public ImagePrint()
     {
          Text = strProgName = "Image Print";

               // Инициализировать объект PrintDocument
               // и стандартные диалоговые окна.

          prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
          setdlg.Document = prndoc;
          prndlg.Document = prndoc;

               // Инициализировать элементы меню.

          Menu.MenuItems[0].Popup += new EventHandler(MenuFileOnPopup);
          Menu.MenuItems[0].MenuItems.Add("-");
```

см. след. стр.

```
            // Элемент Page Setup меню File

        ffiiFileSet" = пен MenuItem("Page Set&up...");
        ffliFileSet. Click += new EventHandler(MenuFileSetupOnClick);
        Menu.MenuItems[0].MenuItems.Add(miFileSet);

            // Элемент Print меню File

        miFilePrint * new MenuItem("&Print...");
        miFilePrint.Click += new EventHandler(MenuFilePrintOnClick);
        miFilePrint.Shortcut = Shortcut.CtrlP;
        Menu.MenuItems[0].MenuItems.Add(miFilePrint);
        Menu.MenuItems[0].MenuItems.Add("-");

            // Элемент Properties меню File

        miFileProps - new MenuItem("Propert&ies...");
        miFileProps.Click += new EventHandler(MenuFilePropsOnClick);
        Menu.MenuItems[0].MenuItems.Add(miFileProps);
    >
    protected override void OnPaint(PaintEventArgs pea)
    {
        if (image != null)
            ScaleImageIsotropically(pea.Graphics, image, ClientRectangle);
    }
    void MenuFileOnPopup ( object obj, EventArgs ea)
    {
        miFileSet.Enabled =
        raiFllePrint. Enabled =
        miFileProps.Enabled = (image != null);
    }
    void MenuFileSetupOnClick(object obj, EventArgs ea)
    {
        setdlg.ShowDialog();
    }
    void MenuFilePrintOnClick(object obj, EventArgs ea)
    {
        if (prndlg.ShowDialog() == DialogResult.OK)
        {
            prndoc.DocumentName = Text;
            prndoc.Print();
        }
    }
    void MenuFilePropsOnClick(object obj, EventArgs ea)
    C
        string str =                    *
            "Size = " + image.Size +
            "\nHorizontal Resolution = " + image.HorizontalResolution +
            "\nVertical Resolution = " + image.VerticalResolution +
            "\nPhysical Dimension - " + image.PhysicalDimension +
```

см. след. стр.

```
                "\nPixel Format = " + image.PixelFormat;
            MessageBox.Show(str, "Image Properties");
        }
        void OnPrintPage(object obj, PrintPageEventArgs ppea)
        {
            Graphics grfx = ppea.Graphics;
            fiectangleF rectf = new RectangleF{
                ppea.MarginBounds.Left -
                (ppea.PageBounds.Width - grfx.VisibleClipBounds.Width) / 2,
                ppea.MarginBounds.Top -
                (ppea.PageBounds.Height - grfx.VisibleClipBounds.Height) / 2,
                ppea.MarginBounds.Width,
                ppea.MarginBounds.Height);

            ScaleImageIsotropically(grfx, image, rectf);
        }
        void ScaleImageIsotropically(Graphics grfx, Image image,
                                     RectangleF rectf)
        {
            SizeF sizef = new SizeF(image.Width / image, HorizontalResolution,
                                    image.Height / image.VerticalResolution);

            float fScale = Math.Min(rectf.Width / sizef.Width,
                                    rectf.Height / sizef.Height);

            sizef.Width *= fScale;
            sizef.Height *= fScale;

            grfx.DrawImage(image, rectf.X + (rectf.Width - sizef.Width } / 2,
                                  rectf.Y + (rectf.Height - sizef.Height) / 2,
                                  sizef.Width, sizef.Height);
        >
}
```

PrintDocument и оба диалоговых окна (*PageSetupDialog* и *PrintDialog*) определяются как поля и инициализируются в конструкторе программы. Конструктор также добавляет в меню элементы Page Setup и Print, а также элемент Properties, показывающий сведения о загруженном изображении.

В обработке элемента меню Page Setup ничего сложного. Метод *MenuFileSetupOnClick* просто вызывает метод *ShowDialog* объекта *PageSetupDialog*, ему даже не приходится проверять значение, возвращаемое этим методом. Элемент меню Print также довольно прост: исполнение *MenuFilePrintOnClick* начинается с вызова *PrintDialog*. Если *PrintDialog* возвращает *DialogResult.OK*, метод *MenuFilePrintOnClick* устанавливает свойство *DocumentName* объекта *PrintDocument* и вызывает метод *Print* (обработчик события *PrintPage* устанавливается во время исполнения конструктора).

Метод *OnPrintPage* сначала рассчитывает размеры прямоугольника области печати с учетом полей при помощи формул из программы PrintWithMargins. Далее он выводит битовую карту страницы максимального размера, позволяющего вписать ее в рассчитанный прямоугольник с сохранением пропорций. Это дела-

ется с помощью слегка модифицированного метода *ScaleImageIsotropically* из программы ImageScaleIsotropic, показанной в главе 11 (я изменил аргумент *Rectangle*, используемый исходной версией метода, на *RectangleF*, чтобы согласовать размеры битовой карты с расчетными размерами прямоугольника). Программа также переопределяет метод *OnPaint* из прежней программы, закладывая в него аналогичную логику для вывода графики.

Предварительный просмотр

Коль скоро приложение поддерживает печать, реализовать в нем функцию предварительного просмотра довольно легко. Это делается при помощи обычного обработчика события *PrintPage*, который в этом случае выводит графику не на принтер, а на поверхность битовой карты, которую затем показывают пользователю. Но, прежде чем продемонстрировать, насколько это легко, посмотрим на «закулисные» действия во время предварительного просмотра. Знание этих подробностей пригодится для реализации собственного подхода к обработке битовых карт для предварительного просмотра.

Для предварительного просмотра ключевым является свойство *PrintController* объекта *PrintDocument*, где по умолчанию находится объект *PrintControllerWithStatusDialog*. Я уже показывал, как поместить в это свойство другой объект, чтобы выбрать иной способ отображения хода печати, отличный от использования диалогового окна статуса.

Более интересного эффекта можно достичь, занеся в свойство *PrintController* объект типа *PreviewPrintController*.

```
PreviewPrintController ppc = new PreviewPrintController();
prndoc.PrintController = ppc;
```

У класса *PreviewPrintController* имеется единственное свойство:

Свойство *PreviewPrintController*

Тип	Свойство	Доступ
boo!	*UseAntiAlias*	Чтение/запись

Остальные свойства объекта *PrintDocument* устанавливают, как при обычной подготовке к печати. Далее следует инициировать печать. Это делается как всегда – вызовом метода *Print* объекта *PrintDocument*.

```
prndoc.Print();
```

Вспомните, что объект «контроллер печати» отвечает за получение объекта *Graphics*, передаваемого объектом *PrintDocument* обработчику события *PrintPage*. Вместо получения объекта *Graphics* для принтера *PreviewPrintController* создает битовую карту для каждой страницы и получает объект *Graphics* для рисования на этой битовой карте. Таким образом, обработчику события *PrintPage* на самом деле передается объект *Graphics*.

После выполнения метода *Print* можно получить доступ к битовым картам страниц. Единственный метод объекта *PreviewPrintController,* не унаследованный им от других объектов, возвращает массив объектов *PreviewPageInfo*:

Метод PreviewPrintController

```
PreviewPageInfo[] GetPreviewPageInfo()
```

У класса *PreviewPageInfo* два свойства:

Свойства PreviewPageInfo

Тип	Свойство	Доступ
(mage	*Image*	Чтение
Size	*PhysicalSize*	Чтение

Значение свойства *image,* выраженное в пикселах, соответствует размеру страницы принтера, выраженному в тех же единицах. Свойство *PhysicalSize* указывает эти же размеры, выраженные в сотых долях дюйма. Теперь у нас есть набор битовых карт, причем каждая соответствует странице печатаемого документа. Эти изображения можно показать в любой момент.

А есть ли более легкий способ предварительного просмотра? Есть. Начните с создания объекта типа *PrintPreviewDialog:*

```
PrintPreviewDialog predlg = new PrintPreviewDialog();
```

PrintPreviewDialog является потомком *Form,* поэтому у него много свойств, методов и событий, большая часть которых нам сейчас неинтересна. Вот некоторые свойства, реализуемые самим объектом *PrtntPreviewDialog:*

Свойства PrintPreviewDialog (выборочно)

Тип	Свойство	Доступ
PrintDocument	*Document*	Чтение/запись
PrintPreviewControl	*PrintPreviewControl*	Чтение
bool	*UseAntiAlias*	Чтение/запись
bool	*HelpButton*	Чтение/запись

Ключевое свойство, которое нужно установить, — *Document.* В него нужно занести тот же объект *PrintDocument,* который применялся для печати и настройки параметров страницы. *PrintPreviewControl* также определяется в пространстве имен *System.Windows.Forms* и предоставляет элементы управления. Они в конечном счете появляются на форме, которая выводит битовые карты с изображениями страниц.

Вот обычный код, выполняющий инициализацию и инициирующий предварительный просмотр:

```
predlg.Document = prndoc;
predlg.ShowDialog();
```

Всю работу выполняет метод *ShowDialog.* Он извлекает из своего свойства *Document* объект *PrintDocument.* заносит в свойство *PrintController* объект *PreviewPrintController,* вызывает метод *Print* объекта *PrintDocument,* после чего показывает форму, отображающую битовые карты страниц при помощи набора элементов управления. При предварительном просмотре документ также можно напечатать. Б этом случае диалоговое окно предварительного просмотра просто использует тот же

самый объект *PrintDocument* с теми же обработчиками событий. Чтобы это сработало, нужно установить свойство *DocumentName* объекта *PrintDocument* перед вызовом метода *ShowDialog*.

Попробуем реализовать в нашей программе — клоне Notepad печать с возможностью настройки параметров страницы и предварительного просмотра. NotepadCloneWithPrinting наследует программе NotepadCloneWithFormat из главы 18.

NotepadCloneWithPrinting.cs

```
//-----------------------------------------------------
// NotepadCloneWithPrinting.cs © 2001 by Charles Petzold
// -                                    ---------   ------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.IO; using System.Windows.Forms;

class NotepadCloneWithPrinting: NotepadCloneWithFormat
{
     PrintDocument       prndoc = new PrintDocument();
     PageSetupDialog    setdlg = new PageSetupDialog();
     PrintPreviewDialog predlg = new PrintPreviewDialog();
     PrintDialog        prndlg = new PrintDialog();
     string              strPrintText;
     int                 iStartPage, iNumPages, iPageNumber;

     public new static void Main()
     {
          System.Threading.Thread.CurrentThread.ApartraentState *
                                        , System.Threading.ApartmentState.STA;

          Application.Run(new NotepadCloneWithPrinting());
     }
     public NotepadCloneWithPrinting()
     {
          strProgName - "Notepad Clone with Printing";
          MakeCaption();

          prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
          setdlg.Document - prndoc;
          predlg.Document = prndoc;
          prndlg.Document - prndoc;

          prndlg.AllowSomePages = true;
          prndlg.PrinterSettings.FromPage - 1;
          prndlg.PrinterSettings.ToPage -
                                   prndlg.PrinterSettings.MaximumPage;
     }
     protected override void MenuFileSetupOnClick(object obj, EventArgs ea)
     {
```

см. след. стр.

```
        setdlg.ShowDialog();
    }
    protected override void MenuFilePreviewOnClick(object obj, EventArgs ea)
    {
        prndoc.DocumentName = Text;    // На тот случай, если документ
                                       // будет напечатан.

        strPrintText = txtbox.Text;
        iStartPage   = 1;
        iNumPages    = prndlg.PrinterSettings.MaximumPage;
        iPageNumber  = 1;

        predlg.ShowDialog();
    }
    protected override void MenuFilePrintOnClick(object obj, EventArgs ea)
    {
        prndlg.AllowSelection = txtbox.SelectionLength > 0;.--

        if (prndlg.ShowDialog() == DialogResult.OK)
        {
            prndoc.DocumentName = Text;

                // Инициализировать ряд важных полей.

            switch (prndlg.PrinterSettings.PrintRange)
            {
            case PrintRange.AllPages:
                strPrintText = txtbox.Text;
                iStartPage   = 1;
                iNumPages    = prndlg.PrinterSettings.MaximumPage;
                break;

            case PrintRange.Selection:
                strPrintText = txtbox.SelectedText;
                iStartPage   = 1;
                iNumPages    = prndlg.PrinterSettings.MaximumPage;
                break;

            case PrintRange.SomePages:
                strPrintText=txtbox.Text;
                iStartPage   = prndlg.PrinterSettings.FromPage;
                iNumPages    = prndlg.PrinterSettings.ToPage -
                               iStartPage + 1;
                break;
            }
              , // И перейти к печати.

            iPageNumber  = 1;
            prndoc.Print();
        }
```

см. след. стр.

```
}
void OnPrintPage(object obj, PrintPageEventArgs ppea)
{
     Graphics     grfx   = ppea.Graphics;
     Font         font   = txtbox.Font;
     float        cyFont = font.GetHeight(grfx);
     StringFormat strfmt = new StringFormat();
     RectangleF   rectfFull, rectfText;
     int          iChars, iLines;

          // Рассчитать значение RectangleF для колонтитулов.

     if (grfx.VisibleClipBounds.X < 0)        // Предварительный просмотр
          rectfFull = ppea.MarginBounds;
     else
          rectfFull = new fiectangleFC
               ppea.MarginBounds.Left - (ppea.PageBounds.Width -
                         grfx.VisibleClipBounds.Width) / 2,
               ppea.MarginBounds.Top - (ppea.PageBounds.Height -
                         grfx.VisibleClipBounds.Height) / 2,
               ppea.MarginBounds.Width, ppea.MarginBounds.Height);

          // Рассчитать RectangleF для текста.

     rectfText = RectangleF.Inflate(rectfFull, 0, -2 * cyFont);

     int iDisplayLines = tint) Math.Floor(rectfText.Height / cyFont);
     rectfText.Height = iDisplayLines * cyFont;

          // Подготовить объект StringFormat для вывода текста
          // в прямоугольной рамке.

     if (txtbox.WordWrap)
     {
          strfmt.Trimming = StringTrimming.Word;
     }
     else
     {
          strfmt.Trimming = StringTrimming.EllipsisCharacter;
          strfmt.FormatFlags |= StringFormatFlags.NoWrap;
     }
          // При печати избранных страниц перейти к первой печатаемой
          // странице.

     while ((iPageNumber < iStartPage) 44 (strPrintText.Length > 0))
     {
          if (txtbox.WordWrap)
               grfx.MeasureString(strPrintText, font, rectfText.Size,
                                  strfmt, out iChars, out iLines);
          else
```

см. след. стр.

```
            iChars = CharsInLines(strPrintText, iDisplayLines);

        strPrintText = strPrintText.Substring(iChars);
        iPageNumber++;
}
        // Отменить задание печати, если текст закончился раньше
        // времени.

if (strPrintText.Length == 0)
{
        ppea.Cancel = true;
        return;
}
        // Вывести текст для этой страницы.

grfx.DrawString(strPrintText, font, Brushes.Black,
                reetfText, strfmt);

        // Получить текст для следующей страницы.

if (txtbox.WordWrap)
        grfx.MeasureString(strPrintText, font, rectfText.Size,
                           strfmt, out iChars, out iLines);
else
        iChars = CharsInLines(strPrintText, iDisplayLines);

strPrintText = strPrintText.Substring(iChars);

        // Очистить объект StringFormat, формирующий колонтитулы.

strfmt = new StringFormat();

        // Вывести имя файла наверху страница,

strfmt.Alignment = StringAlignment.Center;
flrfx.DrawString(FileTitle(), font,   Brushes.Black,
                 rectfFull, strfmt);
        // Вывести внизу страницы ее номер.

strfffIt.LineAlignment = StringAlignment.Far;
grfx.DrawString("Page " + iPageNumber, font, Brushes.Black,
                rectfFull, strfmt);

        // Решить, напечатать ли еще одну страницу.

iPageNumber++;
ppea.HasMorePages = (strPrintText.Length > D) &&
                    (iPageNumber < iStartPage + iNumPages);

    // Снова инициализировать переменные для печати из окна
```

см. след, стр.

```
                // предварительного просмотра.

            if (!ppea.HasMorePages)
            {
                strPrintText = txtbox.Text;
                iStartPage   = 1;
                iNumPages    = prndlg.PrinterSettings.MaximumPage;
                iPageNumber  = 1;
            }
        }
        int CharsInLines(string strPrintText, int iNumLines)
        {
            int index = 0;

            for (int i = 0; i < iNumLines; i++)
            {
                index = 1 + strPrintText.IndexOf('\n', index);

                if (index == 0)
                    return strPrintText.Length;
            }
            return index;
        }
    }
```

Чтобы сделать свою программу хоть немного лучше, чем Notepad, я решил реализовать в диалоговом окне *PrintDialog* возможность печати выбранного диапазона страниц.

В этой программе объект *PrintDialog* и *все три* диалоговых окна определены как поля. Они инициализируются при исполнении конструктора программы. Четыре вспомогательные переменные, необходимые для печати, тоже хранятся в виде полей.

Здесь у метода *MenuFilePrintOnClick* больше возможностей, чем у аналогичного метода в любой из программ, которые мы рассматривали до сих пор. Основное отличие — в возможности печати не только всего документа, но и выбранного диапазона страниц. Так, в переменной *strPrintText* должен быть печатаемый текст. Обычно значение этой переменной получают из *SelectedText*. Проще говоря, весь этот код нужен, чтобы занести в свойства *iStartPage* (первая страница печатаемого диапазона) и *iNumPages* (число страниц для печати) значения, которые зависят от того, что должно быть напечатано: весь документ, диапазон страниц или выделенная часть документа.

Метод *MenuFilePreviewOnClick* (он будет вызван, если щелкнуть элемент меню Print Preview), устанавливает свойства *strPrintText*, *iStartPage*, *iNumPages* и *iPageNumber* как при печати всего документа целиком.

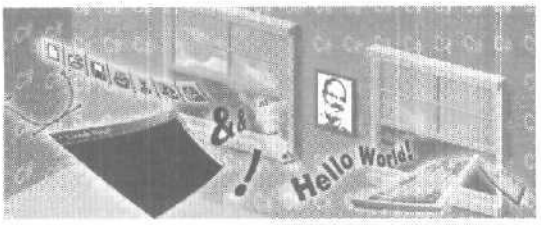
Исполнение обработчика события *PrintPage* начинается с расчета прямоугольника, ограничивающего область вывода графики. В случае обычной печати расчет производится по знакомой вам формуле, использующей значения *MarginBounds*, *PageBounds* и *VisibleClipBounds*. Но если обработчик события вызван методом

PrintPage для предварительного просмотра, то пространство, указанное свойством *VisibleClipBounds* больше, чем битовая карта страницы, а начальная точка визуализации графики расположена в левом верхнем углу этой битовой карты. В этом случае прямоугольник, ограничивающий область вывода графики, просто становится равным *MarginBounds*. Если задан перенос текста по словам, обработчик события *PrintPage* использует механизм переноса из метода *DrawString*. Заметьте; метод *DrawString* корректирует размеры прямоугольника для вывода текста (*rectfText*), чтобы в нем поместилось целое число строк, что позволяет избежать проблемы с обрезанием строк. Аналогично присвоение свойству *Trimming* объекта *StringFormat* значения *StringTrimming.Word* предотвращает обрезание последнего слова на странице. Перегруженная версия метода *MeasureString*, возвращающая число выводимых символов, идеальна для коррекции значения переменной *strPrintText* при подготовке к печати следующей страницы.

Напечатать текст без переноса по словам несколько сложнее. Чтобы добиться правильного вида всех страниц, метод *OnPrintPage* присваивает свойству *Trimming* объекта *StringFormat* значение *StringTrimming.EllipsisCharacter*, а свойству *FormatFlags* ~ значение *StringFormatFlags.NoWrap*. Однако *MeasureString* возвращает именно число выводимых символов, а не число символов, которое было бы показано в отсутствие усечения строк. Мне пришлось написать процедуру *CharsInLines*, которая ищет символы конца строки и соответствующим образом корректирует значение *strPrintText*.

Другой подход к печати текста без переноса заключается в применении свойства *Lines* для получения массива значений *string*, содержащего все строки текста. После этого метод *OnPrintPage* просто по очереди вызывает метод *DrawString* для каждой строки. Но этот подход будет работать лишь при печати всего документа или диапазона страниц, так как у класса *TextBoxBase* нет свойства, возвращающего *выделенный* текст, разбитый на строки.

В диалоговом окне Print Preview есть кнопка, позволяющая распечатать документ, не возвращаясь в приложение. Чтобы обработать этот вариант, по достижении последней страницы метод *OnPrintPage* заново инициализирует поля *strPrintText*, *iStartPage*, *iNumPages* и *iPageNumber*, подготавливая таким образом обычное задание печати.



ГЛАВА

22

Деревья и списки

С программой Проводник (Microsoft Windows Explorer) вы, полагаю, знакомы, В ее клиентской области доминируют два больших и сложных элемента управления. Элемент управления *«дерево»* (tree-view) слева — это иерархический список дисководов и каталогов пользователя. Элемент управления *«список»* (list-view) справа показывает подкаталоги и файлы выбранного каталога в одном их четырех форматов: простой список, таблица из нескольких столбцов, имена файлов с мелкими значками или имена с крупными значками.

В этой главе я рассмотрю реализацию в Windows Forms элементов управления «дерево» и «список». Сложность и непостоянство делают невозможным их всестороннее обсуждение. Но мы, конечно, рассмотрим их достаточно подробно, чтобы вы смогли начать с ними работать.

Однако прежде чем начать, я хотел бы обратить ваше внимание на третий элемент управления Windows Explorer — тонкую вертикальную линию вроде рамки, появляющуюся *между* деревом и списком. Двигая эту линию мышью, можно разделить клиентскую область Windows Explorer между двумя элементами управления так, как вам удобно. Когда один элемент управления становится больше, другой становится меньше. Обеспечивающий это элемент управления называется *разделителем* (splitter),

В мире разделителей

Вы, вероятно, имели дело с разделителями не только в Windows Explorer, но и в других приложениях. В Microsoft Word, например, вы можете выбрать из меню Window (Окно) пункт Split (Разделить) и разбить документ на две области с горизонтальным разделителем. Это позволяет вам работать с одним разделом документа, в то время как другой доступен для просмотра. Кроме того, разделитель позволяет регулировать относительные размеры двух областей просмотра. По

умолчанию клиентская область Microsoft Visual Studio .NET разбита при помощи разделителей на четыре области.

Кроме того, разделители применяются при отображении Web-страниц, использующих *фреймы* (frame), появившиеся в HTML 4.0. В *Спецификации HTML 4-0* (Раздел 16.1) архитектура, основанная на фреймах, объясняется так:

«Фреймы HTML позволяют автору представлять документы в нескольких окнах просмотра, которые могут быть независимыми окнами или подокнами. Несколько окон просмотра позволяют дизайнеру оставить определенную информацию видимой, когда другие окна просмотра прокручиваются или заменяются. Так, в одном и том же окне в одном фрейме отображается неподвижный баннер, во втором — навигационное меню, а в третьем — основной документ, который можно прокручивать или заменять при работе со вторым фреймом*.

Размеры этих фреймов можно зафиксировать, но по умолчанию они регулируются при помощи разделителя. Часто на одной Web-странице присутствуют как горизонтальный, так и вертикальный разделители.

В Windows Forms разделитель реализован в классе *Splitter*, наследуемом от *Control*. Обычно после корректного создания и размещения разделителя можно забыть про этот элемент. Обрабатывать события разделителя требуется довольно редко. Разделитель воздействует на размер других элементов управления аналогично рамке, изменяющей размеры окна. Если ваши элементы управления правильно реагируют на изменение размера окна, они отлично приспособятся к изменению размера при помощи разделителя.

Разделитель всегда связан с *целевым* элементом управления, размер которого он изменяет напрямую, хотя перемещение разделителя может влиять и на другие элементы. Разделитель подключается к целевому элементу посредством механизма стыковки (см. о нем главу 12). Как вы помните, в классе *Control* определено свойство *Dock*, которому можно присвоить один из членов перечисления *DockStyle*:

Перечисление *DockStyle*

Члены	Значение
None	0
Top	1
Bottom	2
Left	3
Right	4
Fill	5

По умолчанию используется стиль *DockStyle.None*. При указании одного из членов таблицы элемент управления разместится вплотную к соответствующей стороне родительского элемента управления и расширится до двух соседних сторон. Так, если *control* — экземпляр любого класса, унаследованного от *Control*, можно выполнить такое присваивание значения свойству *Dock*:

```
control.Dock = DockStyle.Left;
```

Тогда *control* переместится вплотную к левой стороне родительского элемента и изменит размер так, чтобы занять все пространство между верхом и низом роди-

тельского элемента. Всякий раз, когда родительский элемент изменяет свои размеры, *control* соответствующим образом изменяет свои. Класс *Control* выполняет эти магические преобразования при помощи метода *OnResize*. По этой причине (и некоторым другим) при переопределении *OnResize* следует вызывать версию метода из базового класса.

При использовании стиля *DockStyle.Fill* элемент управления занимает всю поверхность предка. Я применял *DockStyle.Fill* в программе AnalogClock из главы 10 в различных версиях программы Notepad из главы 18.

А что, если два элемента управления пристыковать к одной стороне? Или для двух или более элементов применить стиль *DockStyle.Fill*? Или использовать *DockStyle.Fill* для одного элемента и какой-либо член *DockStyle*, не являющийся членом по умолчанию, для другого? Во всех этих случаях поведение определяется z-порядком элементов управления.

Я рассматривал z-порядок в главе 12 и все же приведу его краткий обзор. Элементу управления при помещении на родительский элемент присваивается *дочерний индекс,* который считается с 0. Об элементе с дочерним индексом 0 говорят, что он находится *вверху* z-порядка. Когда элементы управления перекрываются, дочерний, находящийся вверху z-порядка, показывается поверх других элементов и принимает сообщения мыши, когда над ним перемещается курсор. Об элементе управления с самым большим дочерним индексом говорят, что он находится внизу z-порядка.

Программы могут изменять z-порядок элементов управления вызывая методы *BringToFront* и *SendToBack*, реализованные в *Control*, или метод *SetChildIndex*, реализованный в *Control.ControlCollection*. Обычно проще всего сначала создать элементы управления в нужном порядке,

Итак, что случится, если свойство *Dock* у двух или более дочерних элементов имеет значение *DockStyle.Fill*? Дочерний элемент вверху z-порядка, т. е. дочерний элемент, первым добавленный к родительскому элементу и поэтому имеющий минимальный дочерний индекс, появится поверх остальных. Например, рассмотрим код, который вполне может появиться в конструкторе класса, происходящего от *Form*:

```
Panel panel1 = new Panel();
panel1.Parent = this;
panel1.Dock   = DockStyle.Fill;

Panel panel2 = new Panel();
panel2.Parent = this;
panel2.Dock   = DockStyle.Fill;
```

Элемент управления *panell* будет видим, а *panel2* — нет. Общепринятым правилом является то, что только один дочерний элемент может иметь свойство *Dock,* равное *DockStylefill.* (В программе ImageDirectory, которую мы напишем в этой главе, я нарушу это правило. В ней два элемента управления со стилем *DockStyle.Fill*, но в любой момент только у одного из этих элементов свойство *Visible* равно *true.)*

Можно добавить другие элементы управления к родителю перед элементом, у которого свойство *Dock* равно *DockStyle.Fill*, например:

```
Button btn = new Button();
btn.Parent = this;

Panel panel  = new Panel();
panel.Parent = this;
panel.Dock   = DockStyle.Fill;
```

Тогда элемент управления *Button* окажется вверху z-порядка и появится поверх *Panel*. Элемент *Panel* заполнит клиентскую область предка. Будет казаться, что *Button* является дочерним элементом *Panel,* но это не так. Эти элементы просто перекрываются, и приоритет в отношении видимости принадлежит элементу, находящемуся выше в z-порядке.

В следующем случае дело обстоит аналогично:

```
Panel panel1  = new Panel();
panel1.Parent = this;
panel1.Dock   = DockStyle.Left;

Panel panel2  = new Panel();
panel2.Parent = this;
panel2.Dock   = DockStyle.Fill;
```

Заметьте: у первого элемента управления *Panel* свойство *Dock* имеет значение *DockStyle.Left*. Он находится вверху z-порядка, поэтому имеет приоритет. Вторая панель заполнит клиентскую область предка, но ее левая часть закроется панелью *panel1*. Пожалуй, это нежелательно,

А этот вариант гораздо полезнее и (в случае применения разделителя) является общепринятым:

```
Panel panel1  = new Panel{);
panel1.Parent = this;
panel1.Dock   = DockStyle.Fill;

Panel panel2  = new Panel{);
panel2.Parent = this;
panel2.Dock   = DockStyle.Left;
```

Все, что я сделал, — поменял местами два свойства *Dock*. Теперь обе панели полностью видимы. Первая появляется на клиентской области справа, вторая — слева. Если сделать клиентскую область шире или уже, то ***panell*** справа изменит размер, так как ее стилем является *DockStyle.Fill*; *panel2*, находящаяся в клиентской области слева, не изменит размера.

Я уже показывал эту технологию в SimpleStatusBarWithPanel в главе 20. В начале этой программы создавалась *Panel* со свойством *Dock,* равным *DockStyle.Fill*, затем — элемент управления *StatusBar*, в котором по умолчанию используется *DockStyle.Bottom*.

Если изменить последнюю строку показанного выше кода на:

```
panel2.Dock = DockStyle.Right;
```

то panell появится в клиентской области слева, а *panel2* — справа. При изменении размера клиентской области размер *panel1*, которая теперь находится слева,

изменяется соответствующим образом, а размер *panel2* остается таким же. Идеально было бы поместить разделитель между этими двумя панелями.

Что будет, если у двух элементов управления свойства *Dock* имеют одинаковое значение? Например:

```
Panel panel1 = new Panel();
panel1.Parent = this;
panel1.Dock   = DockStyle.Left;

Panel panel2 = new PanelO;
panel2.Parent = this;
panel2.Dock   = DockStyle.Left;
```

При добавлении второго элемента управления к предку элемент вверху z-порядка (*panel1*)выталкивается к центру клиентской области. Элемент управления внизу z-порядка размещается в левом углу предка.

Ниже показан пример двух элементов управления, у которых свойства *Dock* имеют разные значения; кажется, что свойства *Dock* конфликтуют:

```
Panel panel1 = new PanelO;
panel1.Parent = this;
panel1.Dock   = DockStyle.Left;

Panel panel2 = new PanelO;
panel2.Parent = this;
panel2.Dock   = DockStyle.Top;
```

У первой панели свойство *Dock* равно *DockStyle.Left*, у второй — *DockStyle.Top*. Первоначально первая панель прикрепляется к левой стороне клиентской области, а по вертикали растягивается сверху донизу предка. Вторая панель обязательно должна вытолкнуть первую вниз. Поэтому вторая располагается вверху клиентской области и расширяется до ширины клиентской области. Первая ограничивается сверху второй. Обе панели полностью видимы.

Теперь мы готовы приступить к использованию разделителей. Чтобы обобщить рассмотрение, я буду по-прежнему использовать в примерах элементы управления *Panel*. Панели, конечно, могут содержать и другие элементы, в том числе и с прокруткой.

Важно знать, что происходит, когда несколько элементов управления пристыкованы к одной стороне родителя, так как, если вы не будете осторожны, разделитель может оказаться с той же стороны родителя, что не есть хорошо. Целевой элемент управления разделителя — элемент, пристыкованный к тому же углу, что и разделитель, но имеющий меньший z-порядок. Поэтому разделитель обычно создается *до* целевого элемента управления,

Чаще всего разделители размещаются между двумя элементами управления, так что перемещение разделителя увеличивает размер одного и уменьшает размер другого. Но можно использовать разделители еще проще, изменяя размер одного элемента. Ниже разделитель применяется чтобы изменить размер одной панели:

OnePanelWithSplitter.cs

```
//-------------------------------------------------------
// OnePanelWithSplitter.cs 6 2001 fey Charles Petzold
//-------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class OnePanelWithSplitter: Form
{
    public static void Main()
    {
        Application.Run(new OnePanelWithSplitter());
    }
    public OnePanelWithSplitter()
    {
        Text = "One Panel with Splitter";

        Splitter split = new Splitter();
        split.Parent = this;
        split.Dock = DockStyle.Left;

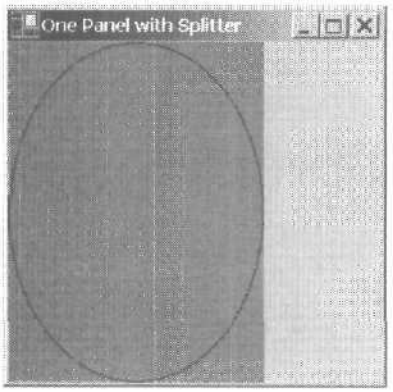
        Panel panel = new Panel();
        panel.Parent = this;
        panel.Dock = DockStyle.Left;
        panel.BackColor = Color.Lime;
        panel.Resize += new EventHandler(PanelOnResize);
        panel.Paint += new PaintEventHandler(PanelOnPaint);
    }

    void PanelOnResize(object obj, EventArgs ea)
    {
        ((Panel) obj).Invalidate();
    }
    void PanelOnPaint(object obj, PaintEventArgs pea)
    {
        Panel panel = (Panel) obj;
        Graphics grfx = pea.Graphics;

        grfx.DrawEllipse(Pens.Black, 0, 0,
                         panel.Width - 1, panel.Height - 1);
    }
}
```

И у разделителя, и у панели свойство *Dock* равно *DockStyle.Left*. Но запомните: элементы управления, находящиеся внизу z-порядка, оказываются ближе к стороне стыковки. Именно поэтому сначала создается разделитель. Панель вытолкнет разделитель из левой стороны родителя, поэтому разделитель окажется справа от панели,

Обычно у панели тот же цвет, что и у ее предка. Поэтому я намеренно изменил ее свойство *BackColor*, чтобы вам не пришлось принимать на веру то, что размер панели действительно изменяется. В этом разделе я использую цвета, чтобы показывать положение элементов управления. В данном случае цвет *lime* означает расположение *слева*. Кроме того, я уже установил обработчики событий *Resize* и *Paint*, так что вы можете дополнительно удостовериться, что видите всю панель, Вот как выглядит эта программа:



В этом примере вы не можете увидеть разделитель, так как у него свойство *BackColor* такое же, как и у предка. Но при перемещении курсора к правой границе панели, курсор превращается в параллельные вертикальные черточки. При этом вы можете настроить размер панели. У разделителей нет интерфейса для работы с клавиатурой. Вертикальные разделители (как в этой программе) используют курсор *Cursors.Vsplit*, а горизонтальные — *Cursors.HSplit*.

Следующая программа показывает две панели, целиком заполняющие клиентскую область. Свойство *Dock* первой панели имеет значение *DockStyle.Fill*, а у разделителя и второй панели это свойство равно *DockStyle.Right*.

TwoPanelsWithSplitter.cs

```
//-------------------------------------------------
// TwoPanelsWithSplitter.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoPanelsWithSplitter: Form
{
     public static void flainO
     {
          Application.Run(new TwoPanelsWithSplitter());
     }
     public TwoPanelsWithSplitter()
     {
          Text = "Two Panels with Splitter";
```

см. след. стр.

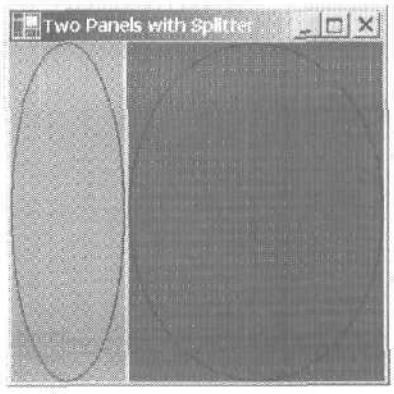
```
        Panel panel1     = new PanelO;
        panel1.Parent    = this;
        panel1.Dock      * DockStyle.Fill;
        panel1.BackColor = Color.Lime;
        panel1.Resize   += new EventHandler(PanelOnResize);
        panel1.Paint    += new PaintEventHandler(PanelOnPaint);

        Splitter split   = new Splitter();
        split.Parent     = this;
        split.Dock       = DockStyle.Right;

        Panel panel2     = new PanelO;
        panel2.Parent    = this;
        panel2.Dock      - DockStyle.Right;
        panel2.BackColor = Color.Red;
        panel2.Resize   += new EventHandler(PanelOnResize);
        panel2.Paint    += new PaintEventHandler(PanelOnPaint);
    }
    void PanelOnResize(object obj, EventArgs sa)
    i
        ((Panel) obj).Invalidate();
    }
    void PanelOnPaint( object obj, Paint Even tArgs pea)
    {
        Panel   panel = (Panel) obj;
        Graphics grfx  = pea.Graphics;

        grfx.DrawEllipse(Pens.Black, 0, 0,
                         panel.Width - 1, panel.Height - 1);
    }
}
```

Экспериментируя с этой программой, вы увидите панель лимонного цвета слева и красного цвета — справа. (Заметьте: lime — left, red — right.) Разделитель стал заметнее, так как у него серый цвет. Вот как выглядит программа после запуска:



Целевым элементом управления разделителя является панель справа, так как и у нее, и у него свойство *Dock* равно *DockStyle.Right*, но разделитель влияет на размер обеих панелей. А если расширить или сузить родительскую форму? Первая панель, расположенная слева, изменит размер, так как использует стиль *DockStyle.Fill*. У такого поведения есть еще одно следствие: если клиентская область станет слишком узкой, левая панель вообще исчезнет, а правая будет «обрезана*.

При создании программы с двумя элементами управления, например панелями с разделителем, стоит подумать, размер какого элемента должен изменяться при изменении размера родителя. Именно он должен создаваться первым со стилем *DockStyle.Fill*.

В следующей программе это правило используется для создания формы с тремя панелями. У центральной панели голубого цвета стиль *DockStyle.Fill*, поэтому при изменении размера клиентской области она изменяет размер,

SplitThreeAcross.cs

```
//--------------------------------------------
// SplitThreeAcross.cs © 2001 by Charles Petzold
//--------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SplitThreeAcross : Form
{
     public static void Main()
     {
          Application.Run(new SplitThreeAcross());
     }
     public SplitThreeAcross()
     {
          Text = "Split Three Across";

          Panel panel1 = new Panel();
          panel1.Parent = this;
          panel1.Dock = DockStyle.Fill;
          panel1.BackColor = Color.Cyan;
          panel1.Resize += new EventHandler(PanelOnResize);
          panel1.Paint += new PaintEventHandler(PanelOnPaint);

          Splitter split1 = new Splitter();
          split1.Parent = this;
          split1.Dock = DockStyle.Left;

          Panel panel2 = new Panel();
          panel2.Parent = this;
          panel2.Dock = DockStyle.Left;
          panel2.BackColor = Color.Lime;
          panel2.Resize += new EventHandler(PanelOnResize);
```

см. след. стр.

```
        panel2.Paint   += new PaintEventHandler(PanelOnPaint);

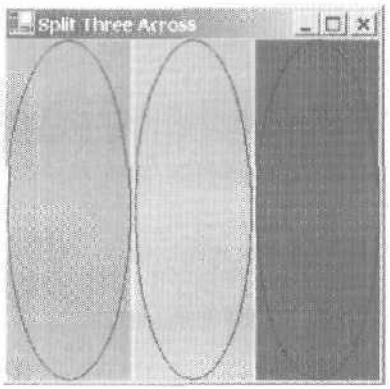
        Splitter split2  = new Splitter();
        split2.Parent    = this;
        split2.Dock      = DockStyle.Right;

        Panel panel3     - new Panel();
        panel3.Parent    = this:
        panel3.Dock      - DockStyle.Right;
        panel3.BackColor = Color.Red;
        panel3.Resize  += new EventHandler(PanelOnResize);
        panel3.Paint   += new PaintEventHandler(PanelOnPaint);

        panel1.Width -
        panel2.Width =
        panel3.Width = ClientSize.Width / 3;
    }
    void PanelOnResize(object obj, EventArgs ea)
    <
        ((Panel) obj).Invalidate();
    }
    void PanelOnPaint(object obj, PaintEventArgs pea)
    i
        Panel     panel = (Panel) obj;
        Graphics grfx = pea.Graphics;

        grfx.DrawEllipse(Pens.Black, 0, 0,
                         panel.Width - 1, panel.Height - 1>;
    }
I
```

И в этом случае, так как центральная панель меняет размер при изменении размера формы, она может исчезнуть, если сделать форму слишком узкой. Если сделать клиентскую область еще уже, правая панель скроется под левой, так как левая панель находится вверху z-порядка. Чтобы гарантировать, что при запуске программы все три панели будут видимы, я присваиваю их ширине значение, равное 1/3 ширины клиентской области.



При таком расположении центральную панель не нужно делать панелью, изменяющей свой размер вместе с клиентской областью. Какой подход окажется более естественным, определяется требованиями конкретного приложения.

Вероятно, лучшим при разработке форм с двумя разделителями является следующий подход. Сначала создается форма с двумя элементами управления и одним разделителем, позволяющим изменять их размеры, а затем — еще два элемента и разделитель как дочерние элементы одного из первоначально созданных элементов управления. Такой подход используется при сочетании горизонтальных и вертикальных разделителей в форме, напоминающей HTML-страницу с фреймами, как, например, в следующей программе.

SplitThreeFrames.cs

```
//-------------------------------------------
// SplitThreeFrames.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SplitThreeFrames: Fora
{
    public static void Nala()
    {
        Application.Run(new SplitThreeFrames());
    }
    public SplitThreeFrames()
    {
        Text = "Split Three Frames";

        Panel panel      - new PanelOr
        panel.Parent     = this;
        panel.Dock       * DockStyle.Fill;

        Splitter split1 = new SplitterO;
        split1.Parent    = this;
        split1.Dock      = DockStyle.Left;

        Panel panel1     = new Panel();
        panel1.Parent    = this;
        panel1.Dock      = DockStyle.Left;
        panel1.BackColor = Color.Lime;
        panel1.Resize   += new EventHandler(PanelOnResize);
        panel1.Paint    += new PaintEventHandler(PanelOnPaint);

        Panel panel2     = new Panel();
        panel2.Parent    * panel;
        panel2.Dock      = DockStyle.Fill;
        panel2.BackColor a Color.Blue;
```

см. след. стр.

```
        panel2.Resize   += new EventHandler(PanelOnResize);
        panel2.Paint    += new PaintEventHandler(PanelOnPaint);

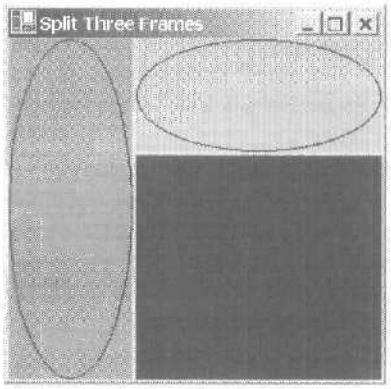
        Splitter split2 = new Splitter();
        split2.Parent    = panel;
        split2.Dock      = DockStyle.Top;

        Panel panel3     = new Panel();
        panel3.Parent    =' panel;
        panel3.Dock      = DockStyle.Top;
        panel3.BackColor = Color.Tan;
        panel3.Resize  += new EventHandler(PanelOnResize);
        panel3.Paint   += new PaintEventHandler(PanelOnPaint);

        panel1.Width  = ClientSize.Width  / 3;
        panel3.Height * ClientSize.Height / 3;
    }
    void PanelOnResize(object obj, EventArgs ea)
    {
        ((Panel) obj).Invalidate();
    }
    void PanelOnPaint(object obj, PaintEventArgs pea)
    {
        Panel    panel = (Panel) obj;
        Graphics grfx  = pea.Graphics;

        grfx.DrawEllipse(Pens.Black, 0, 0,
                         panel.Width - 1, panel.Height - 1);
    }
}
```

В программе SplitThreeFrames сначала создаются две панели (левая и правая), между которыми помещается вертикальный разделитель. Первой панели (названной просто *panel*) присваивается стиль *DockStyle.Fill*. Я не давал ей специального цвета, так как она становится предком других панелей, которые полностью ее закроют. Второй панели (*panel1*, лимонного цвета) присваивается стиль *DockStyle.Left*. В результате получаются две панели и разделитель между ними.



Но это еще не все! Панель *panel* становится предком еще двух панелей и разделителя. Ее первой дочерней панели (*panel2*) присваивается стиль *DockStyle.Fill* и голубой цвет (*blue*). Затем создаются еще один разделитель и панель *panel3* желто-коричневого цвета *(tan)* со свойством *DockStyle.Top*.

В конструкторе устанавливаются начальные размеры левой и верхней панелей.

При изменении размера клиентской области панель внизу справа изменяет размер только потому, что она имеет стиль *DockStyle.Fill* и является дочерним элементом управления другой панели, имеющей стиль *DockStylefill*.

Из свойств класса *Splitter* я пока что использовал лишь *Parent и Dock*. Возможно, самые полезные — такие свойства класса *Splitter* (некоторые унаследованы от *Control):*

Свойства *Splitter* (выборочно)

Тип	Свойство	Доступ
int	*SplitPosition*	Чтение/запись
int	*MinSize*	Чтение/запись
int	*MinExtra*	Чтение/запись
int	*Width*	Чтение/запись
BorderStyle	*BorderStyle*	Чтение/запись
Color	*BackColor*	Чтение/запись

SplitPosition показывает ширину целевого элемента управления для вертикального разделителя и высоту целевого элемента управления для горизонтального разделителя. Если разделитель еще не привязан к целевому элементу управления, свойство будет равно -1. При создании разделителей в конструкторе (как мы обычно и делаем), целевые элементы управления разделителя не назначаются до завершения работы конструктора. Поэтому не используйте в конструкторе *SplitPosition*. Если нужно инициализировать относительные размеры элементов управления, использующих разделители, задавайте значения размерам элементов управления, как показано в программах SplitThreeAcross и SplitThreeFrames.

MinSize показывает минимальную ширину или высоту, до которой можно уменьшить размер целевого элемента управления разделителя. *MinExtra* показывает минимальную высоту или ширину элемента управления, находящегося с другой стороны разделителя. По умолчанию эти свойства равны 25 пикселам. (Действие этих свойств легко увидеть при использовании любой из показанных ранее программ.) Я не советовал бы вам присваивать им нулевые значения, так как пользователя может смутить исчезновение элемента управления при уменьшении его размера. Но им можно присвоить очень маленькие значения, если нужно, чтобы пользователь мог изменять размеры элементов управления в как можно более широких пределах.

Width — это ширина разделителя (по умолчанию 3 пиксела). По умолчанию свойство *BorderStyle* разделителя имеет значение *BorderStyle.None*, при котором разделитель является просто полосой цвета *BackColor*. *BorderStyl.Fixed3D* — это то же самое, что и *BorderStyle.None*. *BorderStyle.FixedSingle* делает разделитель более выступающим, так как его внешние границы становятся черного цвета. Свойство *ForeColor* на разделители не действует.

У класса *Splitter* имеется, кроме событий, определенных в *Control*, еще два события:

События *Splitter*

Событие	Метод	Делегат	Параметр
SplitterMoving	*OnSplitterMoving*	*SplitterEventHandler*	*SplitterEventArgs*
SplitterMoved	*OnSplitterMoved*	*SplitterEventHandler*	*SplitterEventArgs*

Параметром каждого из событий является объект типа *SplitterEventArgs* с такими свойствами:

Свойства *SplitterEventArgs*

Тип	Свойство	Доступ	Описание
int	*X*	Чтение	Положение курсора мыши
int	*Y*	Чтение	Положение курсора мыши
int	*SplitX*	Чтение/запись	Положение разделителя
int	*SplitY*	Чтение/запись	Положение разделителя

Все положения указываются относительно родительского окна разделителя. Свойства *SplitX* и *SplitY* показывают положение левого верхнего угла разделителя в клиентской области. У вертикальных разделителей, *SplitY* равно 0, а у горизонтальных *SplitX* равно 0.

Следующая программа создаст две панели и разделитель. Первоначально каждая из панелей занимает примерно половину клиентской области, но, как вам известно, размер панелей позволяет изменить разделитель. Однако при изменении размера формы размер панелей изменяется пропорционально.

SplitTwoProportional.cs

```
//-----------------------------------------------
// SplitTwoProportional.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SplitTwoProportional : Form
{
     Panel panel2;
     float fProportion = 0.5f;

     public static void Main()
     {
          Application.Run(new SplitTwoProportional());
     }
     public  SplitTwoProportional()
     {
          Text = "Split Two Proportional";
```

см. след. стр.

```
        Panel panel1 = new Panel();
        panel1.Parent = this;
        panel1.Dock = DockStyle.Fill;
        panel1.BackColor = Color.Red;
        panel1.Resize += new EventHandler(PanelOnResize);
        panel1.Paint += new PaintEventHandler(PanelOnPaint);

        Splitter split = new Splitter();
        split.Parent = this;
        split.Dock = DockStyle.Left;
        split.SplitterMoving += new SplitterEventHandler(SplitterOnMoving);

        panel2 = new Panel();
        panel2.Parent = this;
        panel2.Dock = DockStyle.Left;
        panel2.BackColor = Color.Lime;
        panel2.Resize += new EventHandler(PanelOnResize);
        panel2.Paint += new PaintEventHandler(PanelOnPaint);

        OnResize(EventArgs.Empty);
    }
    protected override void OnResize(EventArgs ea)
    {
        base.OnResize(ea);
        panel2.Width = (int) (fProportion * ClientSize.Width);
    }
    void SplitterOnMoving(object obj, SplitterEventArgs sea)
    {
        fProportion = (float) sea.SplitX / ClientSize.Width;
    }
    void PanelOnResize(object obj, EventArgs ea)
    {
        ((Panel) obj).Invalidate();
    }
    void PanelOnPaint(object obj, PaintEventArgs pea)
    {
        Panel panel = (Panel) obj;
        Graphics grfx = pea.Graphics;

        grfx.DrawEllipse(Pens.Black, 0, 0,
                         panel.Width - 1, panel.Height - 1);
    }
}
```

В программе используется экземпляр переменной *fProportion*, равной отношению ширины панели слева к ширине клиентской области. Эта переменная инициализируется значением 0,5 и изменяется при перемещении разделителя пользователем в событии *SplitterOnMoving*. При изменении пользователем размера клиентской области программы метод *OnResize* вычисляет размер панели слева по значению *fProportion* и повой ширине клиента. (Я использовал событие *Splitter-*

Moving, 2 не *SplitterMoved*, так как последнее не работало в соответствии с документацией на момент, когда я писал программу.)

Теперь мы знаем, какой элемент управления находится *между* деревом и списком, и готовы подробно рассмотреть эти два элемента управления.

Деревья и их узлы

Элемент управления *TreeView* часто применяют для показа списка дисков и каталогов вроде того, что отображается слева в Windows Explorer. Однако элементы управления *TreeView* можно применять для показа любой иерархической информации. В Visual Studio .NET дерево служит для отображения проектов и файлов, а в Microsoft Document Explorer (программе, показывающей документацию по программированию в .NET) — пространств имен, классов, членов классов и т. п.

Реализация деревьев в Windows Forms состоит в основном из классов *TreeView*, *TreeNode* и *TreeNodeCollection*. Объект типа *TreeNode* — это одна запись дерева, Объект *TreeNode* связывается со строкой и, возможно, с изображением. В Explorer строкой является имя диска или каталога, а изображения обозначают диски и папки.

Класс *TreeNode* содержит свойство *Nodes*, являющееся набором других объектов *TreeNode;*

Свойства *TreeNode* (выборочно)

Тип	Свойство	Доступ
TreeNodeCollection	*Nodes*	Чтение

Свойство *Nodes* содержит все подузлы узла (их еще называют дочерними узлам и), Класс *TreeNodeCollection* — хорошо знакомый нам вид класса. Он реализует интерфейсы *IList*, *ICollection* и *IEnumerable* и позволяет индексировать набор так же, как и массив:

Свойства *TreeNodeCollection*

Тип	Свойство	Доступ
TreeNode	*[]*	Чтение/запись
int	*Count*	Чтение
bool	*IsReadOnly*	Чтение

Стандартный способ добавления дочерних узлов к существующему узлу — применение методов *Add* и *AddRange* класса *TreeNodeCollection:*

Методы *TreeNodeCollection* (выборочно)

```
TreeNode Add(string strNode)
int Add(TreeNode node)
void AddRange(TreeNode[] anode)
```

Кроме того, *TreeNodeCollection* содержит знакомые методы *Insert*, *Remove*, *Clear* и др. Все узлы, принадлежащие одному объекту *TreeNodeCollection,* иногда называются *братьями* (siblings). Объект *TreeNode,* которому принадлежит набор, называется предком.

Я еще ничего не сказал о самом *TreeView*. По сути это набор объектов *TreeNode* верхнего уровня (корневых объектов). Как и у *TreeNode*, у *TreeView* есть свойство *Nodes*:

Свойства *TreeView* (выборочно)

Тип	Свойство	Доступ
TreeNodeCollection	*Nodes*	Чтение

Это свойство — набор всех корневых объектов *TreeNode*.

Можно сказать, что *TreeView* аналогичен *MamMenu* или *ContextMenu,* тогда как *TreeNode* аналогичен *MenuItem*. (Об объектах меню см. главу 14.) То есть точно так же, как *MamMenu* или *ContextMenu* — набор вложенных объектов *MenuItem*, *TreeView* — набор вложенных элементов *TreeNode*. Однако, как вы помните, все три класса, относящихся к меню, происходят от класса *Menu*. С классами *TreeView* и *TreeNode* дело обстоит иначе: *TreeView* происходит от *Control,* тогда как *TreeNode* порожден *MarshalByRefObject*.

Зная это, мы готовы построить свое первое дерево. Следующая программа демонстрирует иерархию животных, минералов и растений.

SimpleTreeView.cs

```
//---------------------------------------------
// SimpleTreeView.cs © 2001 by Charles Petzold
// ------------------ ..... ---------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleTreeView: Form
{
     public static void Main()
     {
          Application.Run(new SimpleTreeView());
     }
     public SimpleTreeViewO
     {
          Text = "Simple Tree View";

          TreeView tree = new TreeView();
          tree.Parent = this;
          tree.Dock - DockStyle.Fill;

          tree.Nodes.Add("Animal");
          tree.Nodes[0].Nodes.Add("Dog");
          tree.Nodes[0].Nodes[0].Nodes.Add("Poodle");
          tree.Nodes[0].Nodes[0].Nodes.Add("Irish Setter");
          tree.Nodes[0].Nodes[0].Nodes.Add("German Shepherd");
          tree.Nodes[0].Nodes.Add("Cat");
          tree.Nodes[0].Nodes[1].Nodes.Add("Calico");
```

см. след. стр.

```
            tree.Nodes[0].Nodes[1].Nodes.Add("Siamese");
            tree.Nodes[0].Nodes.Add("Primate");
            tree.Nodes[0].Nodes[2].Nodes.Add("Chimpanzee");
            tree.Nodes[0].Nodes[2].Nodes.Add("Ape");
            tree.Nodes[0].Nodes[2].Nodes.Add("Human");
            tree.Nodes.Add("Mineral");
            tree.Nodes[1].Nodes.Add("Calcium");
            tree.Nodes[1].Nodes.Add("Zinc");
            tree.Nodes[1].Nodes.Add("Iron");
            tree.Nodes.Add("Vegetable");
            tree.Nodes[2].Nodes.Add("Carrot");
            tree.Nodes[2].Nodes.Add("Asparagus");
            tree.Nodes[2].Nodes.Add("Broccoli");
        }
    }
```

Весь элемент *TreeView* создается в конструкторе 20 вызовами метода *Add* класса *TreeNodeCollection*.Три из пих выполняются для свойства *Nodes* объекта *TreeView* и, следовательно, создают корневые узлы:

```
tree.Nodes.Add("Animal");
⋮
tree.Nodes.Add("Mineral");
⋮
tree.Nodes.Add("Vegetable");
```

Хотя программа не ссылается ни на один объект *TreeNode,* эти объекты здесь присутствуют. Каждый из методов *Add* создает объект *TreeNode.* Второй вызов *Add* выглядит так;

```
tree.Nodes[0].Nodes.Add("Dog");
```

Первая часть этого оператора (*tree.Nodes[0]*ссылается на первый объект *TreeNode* набора *TreeView*т. е. на «Animal». Второе свойство *Nodes* — набор дочерних объектов узла «Animal», в которую добавляется узел «Dog». Аналогично следующий оператор добавляет дочерний узел узла «Dog»:

```
tree.Nodes[0].Nodes[0].Nodes.Add("Poodle");
```

Вот как выглядит программа, когда раскрыты некоторые узлы дерева:



Возможно, стоит поэкспериментировать с программой, чтобы получить небольшой опыт работы со стандартным пользовательским интерфейсом, реализованным в элементах управления *TreeView*. Клавиши-стрелки «вверх» и «вниз» позволяют изменить выделенный элемент, показанный инверсным цветом. Если слева от элемента стоит знак «плюс», то, нажав «стрелку-вправо», можно просмотреть дочерние узлы. «Стрелка-влево» выполняет двойную функцию. Если выбранный узел не имеет подузлов, то нажатие этой клавиши вызывает переход на родительский узел. Если выбранный узел уже раскрыт (в этом случае слева от него стоит «минус»), то «стрелка-влево» сворачивает узел. Кроме того, конечно, можно щелкать «плюсы» и «минусы» для раскрытия и сворачивания узлов.

По умолчанию полосы прокрутки показываются в случае необходимости. Если их не нужно показывать, присвойте *false* свойству *Scrollable* класса *TreeView*.

Хотя SimpleTreeView создает весь элемент управления *TreeView* прямо в конструкторе, часто бывает, что в дальнейшем программе приходится во время выполнения изменять содержимое *TreeView*. Можно отсортировать элементы дерева, присвоив *true* свойству:

Свойства *TreeView* (выборочно)

Тип	Свойство	Доступ
bool	*Sorted*	Чтение/запись

Независимо от того, используется ли сортировка элементов, чтобы предотвратить возникновение проблем с производительностью, вызывайте до и после выполнения любой последовательности операторов, влияющей на многие узлы дерева, методы:

Методы ***TreeView*** (выборочно)

```
void BeginUpdate()
void EndUpdate()
```

Изображения в деревьях

Как видно из программы SimpleTreeView, чтобы воспользоваться «плюсом» и «минусом», ничего делать не нужно. Однако показ этих знаков можно и подавить. Следующие свойства имеют по умолчанию значение *true:*

Свойства *TreeView* (выборочно)

Тип	Свойство	Доступ
bool	*ShowPlusMinus*	Чтение/запись
bool	*ShowLines*	Чтение/запись
bool	*ShowRootLines*	Чтение/запись

Если *ShowLines* равно *false*, все линии, обычно соединяющие узлы, не показываются, Если *ShowRootLines* равно *false*, корневые узлы показываются без линий, «плюсов» и «минусов». Остальные элементы показываются, как обычно.

В элементах управления *TreeView* часто показываются маленькие картинки слева от каждого узла. При раскрытии узла эти картинки могут изменяться. Так, в Windows Explorer отображается закрытая папка для свернутого узла каталога и раскрытая — для раскрытого,

Изображения элемента управления *TreeView* хранятся в объекте *ImageList.* относящемся ко всему *TreeView:*

Свойства *TreeView* (выборочно)

Тип	Свойство	Доступ
ImageList	*ImageList*	Чтение/запись
ml	*ImageIndex*	Чтение/запись
int	*SelectedImageIndex*	Чтение/запись

Свойство *ImageIndex* определяет изображение по умолчанию, показываемое для невыбранного узла, а *SelectedImageIndex*— для выбранного.

Вообще можно указывать индексы изображения для любого объекта *TreeNode:*

Свойства *TreeNode* (выборочно)

Тип	Свойство	Доступ
int	*ImageIndex*	Чтение/запись
int	*SelectedImageIndex*	Чтение/запись

Однако у *TreeNode* нет собственного свойства *ImageList.* Индексы относятся к свойству *ImageList* элемента управления *TreeVieiv,* которому принадлежит объект *TreeNode,*

Кроме того, можно указывать индексы изображений при создании объекта *TreeNode.* Вот полный список конструкторов *TreeNode:*

Конструкторы *TreeNode*

```
TreeNode()
TreeNode(string strNode)
TreeNode(string strNode, TreeNode[] anodes)
TreeNode(string strNode, int indexImage, int indexImageSelected)
TreeNode(string strNode, int indexImage, int indexImageSelected,
         TreeNode[] anodes)
```

В программе SimpleTreeView я вызывал метод *Add* класса *TreeNodeCollection* со строковым параметром. Есть еще одна версия *Add c* параметром *TreeNode.* Так что можно сначала создать элемент *TreeNode,* а затем добавить его в объект *TreeNode-Collection.* Два показанных выше конструктора *TreeNode* позволяют указывать массив дочерних элементов *TreeNode.* Эти конструкторы позволяют строить иерархию *TreeView* от узлов самого нижнего уровня до корня.

События дерева

Класс *TreeNode* не содержит собственных событий. Однако в классе *TreeView,* помимо событий, унаследованных от *Control,* определено еще 11 событий. Вот 6 самых важных:

События *TreeView* (выборочно)

Событие	Метод	Делегат	Параметр
BeforeExpand	*OnBeforeExpand*	*TreeViewCancelEventHandler*	*TreeViewCancelEventArgs*
BeforeCollapse	*OnBeforeCollapse*	*TreeViewCancelEventHandler*	*TreeViewCancelEventArgs*
BeforeSelect	*OnBeforeSelect*	*TreeViewCancelEventHandler*	*TreeViewCancelEventArgs*
AfterExpand	*OnAfterExpand*	*TreeViewEventHandler*	*TreeViewEventArgs*
AfterCollapse	*OnAfterCollapse*	*TreeViewEventHandler*	*TreeViewEventArgs*
AfterSelect	*OnAfterSelect*	*TreeViewEventHandler*	*TreeViewEventArgs*

Эти события происходят, когда пользователь или программа раскрывают, сворачивают или развертывают узел. Можно видеть, что эти события происходят парами. События, имена которых начинаются со слова *Before*, происходят до того, как *TreeView* выполнит операцию. Объект *TreeViewCancelEventArgs*, связанный с этим событием, имеет свойства:

Свойства *TreeViewCancelEventArgs*

Тип	Свойство	Доступ
TreeNode	*Node*	Чтение
TreeViewAction	*Action*	Чтение
bool	*Cancel*	Чтение/запись

Свойство *Node* — это объект *TreeNode*, который пользователь пытается раскрыть, свернуть или выбрать. *Свойство Action* является членом перечисления:

Перечисление *TreeView Action*

Член	Значение
Unknown	0
ByKeyboard	1
ByMouse	2
Collapse	3
Expand	4

Если почему-либо программа решает, что выполнять операцию нельзя, она присваивает *true* свойству *Cancel*, унаследованному от *CancelEventArgs*.

В противном случае элемент управления *TreeView* раскроет, свернет или выберет узел, и произойдет событие, имя которого начинается со слова *After*. Сопутствующий этому событию объект *TreeViewEventArgs* имеет свойства:

Свойства *TreeViewEventArgs*

Тип	Свойство	Доступ
TreeNode	*Node*	Чтение
TreeViewAction	*Action*	Чтение

Об остальных событиях, реализованных в *TreeView*, я умолчу. События *BeforeLabelEdit* и *AfterLabelEdit* происходят, только когда свойство *LabelEdit* имеет значение *true*. При этом пользователь может редактировать текст узла дерева. События *BeforeCheck* и *AfterCheck* происходят, только когда *true* задано свойству *Check-*

Boxes, показывающему, что рядом с узлами дерева помещаются флажки, которые пользователь может устанавливать и сбрасывать. У *TreeNode* имеется свойство *Checked,* означающее, что флажок узла установлен. Событие *ItemDrag* происходит при буксировке объекта в элемент управления *TreeView.*

TreeView наследует многие методы, свойства и события от *Control.* Так, если нужно реализовать контекстное меню, элементы которого зависят о того, какой узел щелкнули правой кнопкой мыши, можно установить обработчик события *MouseDown* и передать координаты мыши методу *GetNodeAt* класса *TreeView.*

Перемещение по узлам

При вызове обработчика события *TreeView* обработчик события обычно должен выполнять определенные действия в зависимости от конкретного узла, который раскрывается, сворачивается или выбирается. Узлы идентифицируются несколькими способами. Вот некоторые полезные свойства:

Свойства *TreeNode* (выборочно)

Тип	Свойство	Доступ
TreeView	*TreeView*	Чтение
int	*Index*	Чтение
string	*Text*	Чтение/запись
object	*Tag*	Чтение/запись

Свойство *TreeView* указывает элемент управления *TreeView,* к которому относится данный объект *TreeNode, Index* — это индекс узла в коллекции, состоящей из узла и его братьев, *Text* — текст, отображаемый узлом, а *Tag* позволяет присоединять к узлу произвольную информацию для идентификационных или других целей

Кроме того, класс *TreeNode* содержит несколько неизменяемых свойств, позволяющих программе перемещаться по узлам:

Свойства *TreeNode* (выборочно)

Тип	Свойство	Доступ
TreeNode	*Parent*	Чтение
TreeNode	*FirstNode*	Чтение
TreeNode	*LastNode*	Чтение
TreeNode	*NextNode*	Чтение
TreeNode	*PrevNode*	Чтение
TreeNode	*NextVisibleNode*	Чтение
TreeNode	*PrevVisibleNode*	Чтение

Свойство *Parent* указывает на родительский узел, а свойства *FirstNode* и *LastNode* — на дочерние узлы. (К дочерним узлам можно обращаться и через свойство *Nodes* узла.) Свойства *NextNode* и *PrevNode* — это узлы-братья, а *NextVisibleNode* и *PrevVisibleNode* могут быть братьями, детьми или предками — это следующий или предыдущий узел, который выберется при нажатии стрелки «вверх» или «вниз».

Класс *TreeNode* содержит два свойства, позволяющих программе определить, раскрыт, свернут или выбран узел:

Свойства *TreeNode* (выборочно)

Тип	Свойство	Доступ
bool	*IsExpanded*	Чтение
bool	*IsSelected*	Чтение

Можно раскрывать или сворачивать узлы программно, без участия пользователя:

Методы *TreeNode* (выборочно)

```
void Expand()
void ExpandAll()
void Collapse()
void Toggle()
```

Метод *ExpandAll* раскрывает все подузлы узла, для которого вызывается этот метод, их подузлы, подузлы подузлов и т. д. Раскрыть/свернуть все дерево позволяют методы *TreeView*:

Методы *TreeView* (выборочно)

```
void ExpandAll()
void CollapseAll()
```

Следующее свойство позволяет определить, какой узел *TreeView* выбран, или выбрать узел:

Свойства *TreeView* (выборочно)

Тип	Свойство	Доступ
TreeNode	*SelectedNode*	Чтение/запись

Имеется еще один, порой удобный способ идентификации конкретного узла, в котором используется свойство *TreeNode:*

Свойства *TreeNode* (выборочно)

Тип	Свойство	Доступ
string	*FullPath*	Чтение

Для любого узла *FullPatb* содержит строку, получаемую путем конкатенации текста всех родительских узлов, начиная с корневого уровня. Текст узлов разделяется СИМВОЛОМ:

Свойства *TreeNode* (выборочно)

Тип	Свойство	Доступ
string	*PathSeparator*	Чтение/запись

По умолчанию *PathSeparator* — обратная косая черта. Например, в программе SimpleTreeView свойство *FullPatb* для узла «Siamese» равно «Animal\Cats\Siamese».

Свойство *FullPath* идеально подходит при использовании элемента управления *TreeView* для отображения дисков и каталогов. Этим мы и займемся.

Дерево каталогов

Как известно, показ дисков и каталогов — типичная задача, решаемая с помощью *TreeView*. Можно было бы надеяться, что в Windows Forms есть класс, унаследованный от *TreeView* и реализующий стандартное дерево каталогов. Но даже если бы он был, все равно его использование не доставило бы такого удовольствия, как разработка собственного класса!

Следующий класс *DirectoryTreeView* является наследником *TreeView* и используется в следующих двух программах и в еще одной программе (ExplorerLike) в конце главы. В классе применяются кое-какие классы ввода-вывода, описанные в приложении А.

DirectoryTreeView.cs

```
//-----------------------------------------
// DirectoryTreeView.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class DirectoryTreeView: TreeView
{
     public DirectoryTreeView()
     {
          // Выделяем дополнительное пространство для имен каталогов.

          Width *= 2;

          // Получаем изображения для дерева,

          ImageList = new ImageList();
          ImageList.Images.Add(new Bitmap(GetType(), "35FLOPPY.BMP"));
          ImageList.Images.Add(new Bitmap(GetType(), "CLSDFOLD.BMP"));
          ImageList.Images.Add(new Bitmap(GetType(), "OPENFOLD.BMP"));

          // Строим дерево.

          RefreshTree();
     }
     public void RefreshTree()
     {
          // Отключаем визуальное изменение и строим дерево.

          BeginUpdate();
          Nodes.Clear();

          // Делаем диски корневыми узлами.
```

см. след. стр.

```
        string[] astrDrives = Directory.GetLogicalDrives();

        foreach (string str in astrDrives)
        {
            TreeNode tnDrive = new TreeNode(str, 0, 0);
            Nodes.Add(tnDrive);
            AddDirectories(tnDrive);

            if (str == "C:\\")
                SelectedNode = tnDrive;
        }
        EndUpdate();
    }
    void AddDirectories(TreeNode tn)
    {
        tn.Nodes.Clear();

        string          strPath = tn.FullPath;
        DirectoryInfo   dirlnfo = new DirectoryInfo(strPath);
        DirectoryInfo[] adirinfo;

        try
        {
            adirinfo - dirinfo.GetDirectories();
        }
        catch
        {
            return;
        }

        foreach (Directorylnfo di in adirinfo)
        {
            TreeHode tnDir = new TreeNode(di.Name, 1, 2);
            tn.Nodes.Add(tnDir);

            // Мы могли бы заполнить все дерево одним оператором;
            //        AddDirectories(tnDir);
            // но это будет работать слишком медленно. Попробуйте!
        }
    }
    protected override void OnBeforeExpand(TreeViewCancelEventArgs tvcea)
    {
        base.OnBeforeExpand(tvcea);

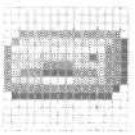
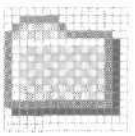
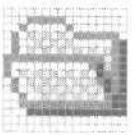
        BeginUpdate();

        foreach (TreeNode tn in tvcea.Node.Nodes)
            AddDirectories(tn);
```

см. след. стр.

```
            EndUpdate();
        }
    }
```

35Floppy.bmp **Cldsfold.bmp** **Openfold.bmp**

DirectoryTreeView использует три небольших картинки, которые я скопировал из набора, поставляемого с Visual Studio .NET. По умолчанию они находятся Е каталоге Program Files\Microsoft Visual Studio .NET\Common7\Graphics\Bitmaps\Outline\NoMask. Я предпочел бы показывать разные картинки для разных типов дисков (дисковод, жесткий диск, CD-ROM и т. д.). но классы Windows Forms не позволяют определить тип диска. При построении объекта *ImageList* в конструкторе предполагается, что пространство имен ресурса является пустой строкой.

В классе *DirectoryTreeView* определен один открытый метод — *RejreshTree*. Программы, использующие класс, могут вызывать его для полного обновления структуры каталогов. [Как вам, наверное, известно, программы, использующие деревья для показа каталогов, обычно содержат пункт меню Refresh (Обновить).] Кроме того, *RefreshTree* вызывается в конструкторе для построения дерева.

RejreshTree получает строковые представления имеющихся в системе дисков, вызывая статический метод *Directory.GetLogicalDrives*, который возвращает массив строк и обычно начинается с «A:\», «C:\» и т. д. Эти строки становятся корневыми узлами. Для каждого диска *RefreshTree* вызывает *AddDirectories*.

Метод *AddDirectories* получает в качестве параметра *TreeNode* и создает дочерние узлы, содержащие имена подкаталогов. В нем для создания объекта *DirectoryInfo* применяется замечательное свойство *FullPath* класса *TreeNode*. Затем вызывается метод *GetDirectories* класса *Directorylnfo,* чтобы получить массив объектов *Directorylnfo,* используемых при создании дочерних узлов.

Directorylnfo может вызвать исключение. Это произойдет, например, если в дисководе нет дискеты или доступ к каким-то каталогам запрещен. Поэтому метод вызывается в блоке *try*. К сожалению, если в дисководе нет дискеты (для диска А эта ситуация возникает часто), то *GetDirectories*, кроме того, показывает раздражающее сообщение об этой проблеме, *перед* генерацией исключения. (Закрыть это информационное окно можно, нажав клавишу Cancel или Continue.) Это окно показывается, даже если *GetDirectories* вызывается консольным приложением! Очевидно, это упущение *в* проектировании или ошибка в реализации метода *GetDirectories*. Как бы то ни было, пока ошибку не исправят, появления информационного окна не избежать. Если при разработке приложения это критично, вместо этого метода используйте функции Win32 API *FindFirstFile* и *FindNextFile*.

Я хорошо понимаю, что любой программист, столкнувшийся с задачей построения дерева каталогов, сразу же подумает *о рекурсивных функциях*. Действительно, путем рекурсивного вызова *AddDirectories* можно построить все дерево ката-

логов за раз. Я даже поместил оператор (правда, в закомментированном виде), рекурсивно вызывающий *AddDirectories*. Вы можете попробовать удалить две косых черты и увидеть, почему я отказался от такого подхода: построение дерева потребует слишком много времени. Гораздо эффективнее вызывать *AddDirectories* только при необходимости.

А зачем вообще вызывать *AddDirectories* в методе *RefreshTree*? Сначала в дереве должны показываться только диски. Вызов *AddDirectories* для каждого диска может показаться необязательным. Однако диски, содержащие каталоги, должны показываться со знаком «плюс», позволяющим пользователю раскрыть узел. Единственный способ добиться показа «плюса» — добавить подузлы. Поэтому даже если первоначально показываются только диски, корневые подкаталоги каждого диска также добавляются в дерево.

Кроме того, в классе *DirectoryTreeView* переопределяется метод *OnBeforeExpand* класса *TreeView*. Первый раз этот метод вызывается, когда пользователь раскрывает один из узлов-дисков. Однако *OnBeforeExpand* не должен формировать набор подузлов для диска, так как она уже существует. Вместо этого он должен построить набор дочерних узлов для новых показываемых узлов — это нужно только для того, чтобы в *TreeView* показывался «плюс» для каталогов, содержащих подкаталоги.

Следующая программа простым образом использует DirectoryTreeView. Она создает справа элемент управления *Panel*, слева — *DirectoryTreeView* и помещает между ними *Splitter*. Она устанавливает обработчик события *AfterSelect*, унаследованного *DirectoryTreeView* от *TreeView*, и показывает список файлов этого каталога в панели.

DirectoriesAndFiles.cs

```
//---------------------------------------------------
// DirectoriesAndFiles.cs $ 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class DirectoriesAndFiles: Form
{
     DirectoryTreeView dirtree;
     Panel             panel;
     TreeNode          tnSelect;

     public static void Main()
     {
          Application.Run(new DirectoriesAndFiles());
     }
     public DirectoriesAndFiles()
     {
          Text = "Directories and Files";
```

см. след. стр.

```
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;

        panel = new Panel();
        panel.Parent * this;
        panel.Dock = DockStyle.Fill;
        panel.Paint += new PaintEventHandler(PanelOnPaint);

        Splitter split * new Splitter();
        split.Parent = this;
        split.Dock = DockStyle.Left;
        split.BackColor = SystemColors.Control;

        dirtree - new DirectoryTreeView();
        dirtree.Parent * this;
        dirtree.Dock = DockStyle.Left;
        dirtree.AfterSelect +=
                new TreeViewEventHandler(DirectoryTreeViewOnAfterSelect);

            // Создаем меню, состоящее из одного пункта.  .

        Menu - new MainMenu();
        Menu.MenuItems.Add("View");

        MenuItem mi * new MenuItem("Refresh",
                        new EventHandler(MenuOnRefresh), Shortcut.F5);
        Menu.MenuItems[0].MenuItems.Add(mi);
    }
    void DirectoryTreeViewOnAfterSelect(object obj, TreeViewEventArgs tvea)
    {
        tnSelect = tvea.Node;
        panel.Invalidate();
    }
    void PanelOnPaint(object obj, PaintEventArgs pea)
    {
        if (tnSelect == null)
            return;

        Panel         panel     = (Panel) obj;
        Graphics      grfx      = pea.Graphics;
        DirectoryInfo dirinfo   = new DirectoryInfo(tnSelect.FullPath);
        FileInfo[]    afileinfo;
        Brush         brush     = new SolidBrush(panel.ForeColor);
        int           y         = 0;

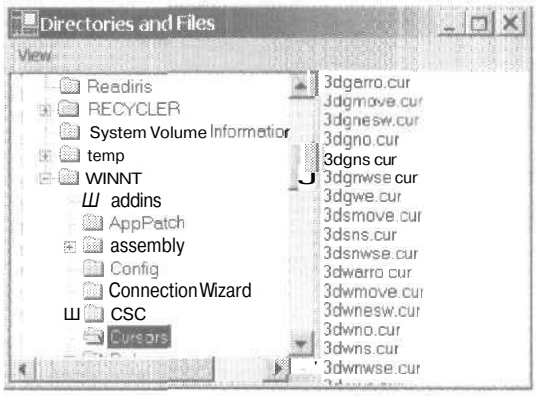
        try
        {
            afileinfo   =  rfirinfo.GetFiles<);
        I
        catch
```

см. след. стр.

```
            {
                return;
            }

            foreach (FileInfo fileinfo in afileinfo)
            {
                flrfx.OrawString(fiieinfQ.Nam8, Font, brush, 0, y);
                y += Font.Height;
            }
        }
        void MenuOnRefresh(object obj, EventArgs ea)
        {
            dirtree.RefreshTree();
        }
    }
```

Так как это всего лишь демонстрационная программа, список файлов состоит только из одного столбца, так что может произойти «обрезание» списка. Вот как выглядит один из подкаталогов моего каталога WINNT:



Кроме того, программа содержит меню View, состоящее из одного пункта Refresh. При выборе этого пункта выполняется обновление дерева каталогов, для чего вызывается метод *RefresbTree* класса *DirectoryTreeView*.

Показ изображений

В главе 23 я подробно рассмотрю метафайлы, являющиеся двоичными наборами команд рисования графики, описывающих изображение. Для подготовки к изучению этой главы я хотел бы посмотреть свою библиотеку изображений в формате метафайлов. Обычно для просмотра каталогов с большим количеством изображений я использую специальную бесплатную программу. Она показывает слева дерево, а справа — пиктограммы. Можно щелкнуть пиктограмму для просмотра полноразмерного изображения. Эта программа отлично работает со многими форматами растровых изображений, но совершенно не способна читать метафайлы. Последние версии Windows Explorer показывают пиктограммы растровых изображений и метафайлов, но Windows Explorer требуется внешняя программа для показа полноразмерных изображений.

В Windows Forms и *Bitmap*, и *Metafile* происходят от *Image*. Метафайлы можно читать с диска, как растровые изображения, используя статический метод *ImageFromFile*, и их можно также легко показывать с помощью *DrawImage*. В программу Windows Forms, загружающую и показывающую растровые изображения, ничего не стоит добавить поддержку метафайлов.

Поставим перед собой простую задачу: написать программу ImageDirectory, которая выводит слева элемент управления *TreeView*, содержащий каталоги, а с. права — набор пиктограмм, который отображает все файлы изображений (растровых изображений *и* метафайлов), находящиеся в выбранном каталоге. По щелчку пиктограммы изображение должно показываться с размером, увеличенным до размеров формы.

Мы уже написали значительную часть этой программы — элемент управления *DirectoryTreeView*. Другой ее частью будет клиентская область, состоящая из элемента управления *Panel*. Пиктограммы будут элементами управления *Button*, показывающими изображение, уменьшенное до размера кнопки. Вот как выглядит элемент управления *ImagePanel*, унаследованный от *Panel* и решающий эту задачу,

ImagePanel.cs

```
//---------------------------------------
// ImagePanel.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class ImagePanel: Panel
{
     const int cxButton = 100, cyButton = 100;          // Размер кнопки
                                                        // с изображением

     Button    fetnCHeked;
     ToolTip   tooltip = new ToolTip();
     Timer     timer = new Timer();

          // поля для события Tick таймера

     string[]  astrFileNames;
     int       i, x, y;

                                                        // Открытое событие
     public event EventHandler ImageClicked;

                                                        // Конструктор
     public ImagePanel()
     {
          AutoScroll = true;

          timer.Interval = 1;
          timer.Tick += new EventHandler(TimerOnTick);
```

см. след. стр.

```
}
                                                    // Открытые свойства
public Control ClickedControl
{
    get { return btnClicked; } ;:
}
public Image ClickedImage
{
    get
    {
        try
        {
            return Image.FromFile((string) btnClicked.Tag);
        }
        catch
        {
            return null;
        }
    }
}
                                                    // Открытый метод
public void ShowImages(string strDirectory)
{
    Controls.Clear();
    tooltip.flemoveAIK);

    try
    {
        astrFileNames = Directory.GetFiles(strDirectory);
    }
    catch
    {
        return;
    }

    i a x я у а 0;

    timer.Start();
}
                                                    // Обработчики событий
void TimerOnTick(object obj, EventArgs ea)
{
    Image image;

    if (i == astrFileNames.Length)
    C
        timer.Stop();
        return;
    |"
    try
```

см. след. стр.

```
        {
            image = Image.FromFile(astrFileNames[i]);
        }
        catch
        {
            i++;
            return;
        >
        int cxImage = image.Width;
        int cyImage = image.Height;

            // Уменьшение размера картинки для кнопки.

        SizeF sizef - new SizeF(cxImage / image.HorizontalResolution,
                                cyImage / image.VerticalResolution);

        float fSeale = Math.Min(cxButton / sizef.Width, ••
                                cyButton / sizef.Height);
        sizef.Width *= fSeale;
        sizef.Height *= fSeale;
        Size size = Size.Ceiling(sizef);
        Bitmap bitmap = new Bitmap(image, size);
        Image.Dispose();

            // Создаем кнопку и помещаем ее на панель.

        Button btn    = new Button();
        btn.Image     - bitmap;
        btn.Location = new Point(x, y) + (Size) AutoScrollPosition;
        btn.Size      = new Size(cxButton, cyButton);
        btn.Tag       = astrFileNames[i];
        btn.Click   += new EventHandler(ButtonOnClick);
        Controls.Add(btn);

            // Назначаем подсказку для кнопки.

        tooltip.SetToolTip(btn, String.Format("{0}\n{1}x{2}",
                                    Path.GetFileName(astrFileNames[i ] ) ,
                                    cxImage, cyImage));

            // Вычисляем i, x и y для следующего изображения.

        AdjustXY(ref x, ref y);
        i++;
    }
    void ButtonOnClick(object obj, EventArgs ea)
    {
        btnClicked = (Button) obj;

        if (ImageClicked != null)
```

см. след. < mp.

```
            ImageClicked(this, EventArgs.Empty);
        }
        protected override void OnResize(EventArgs ea)
        {
            base.OnResize(ea);

            AutoScrollPosition = Point.Empty;
            int x * 0, y = 0;

            foreach (Control cntl in Controls)
            {
                cntl.Location = new Point(x, y) + (Size) AutoSc rollPosition;
                AdjustXY(ref x,  ref y);
            }
        }

        void AdjustXY(ref int x, ref int y)
        {
            y += cyButton;

            if (y+ cyButton > Height -
                              SystemInformation.HorizontalScrollBarHeight)
            {
                y = 0;
                x += cxButton;
            }
        }
}
```

Конструктор *ImagePanel* присваивает свойству *AutoScroll* значение *true*. Если кнопки не помещаются на панели, нужны полосы прокрутки, чтобы пользователь мог получить доступ к кнопкам.

В *ImagePanel* реализован открытый метод *ShowImages*, единственным параметром которого является имя каталога. *Sbowlmages* получает массив всех файлов этого каталога, загружает объект *Image* для файлов каталога, для которых при вызове *ImageFromFile* не сгенерировалось исключение, создает растровое изображение, содержащее изображение из файла, размер которого уменьшен до размера кнопки, создает саму кнопку и, кроме того, подсказку, состоящую из имени изображения и его размера в пикселах.

Вообще-то так было только в первой версии программы. Выяснилось, что для каталогов, содержащих много больших изображений, программа слишком долго выполняет описанную выше работу. Я решил распределить эту работу при помощи объекта *Timer*. (Это простейшая форма многозадачности, не требующая нескольких потоков.) Объект *Timer* создается как поле объекта *ImagePanel* и в конструкторе *ImagePanel* ему присваивается интервал срабатывания 1 миллисекунда. Метод *Sbowlmages* после получения массива имен файлов инициализирует несколько переменных (индекс массива имен файлов и координаты *x и y* кнопок) и запускает таймер.

В обработчике события *Tick* вызывается *ImageFromFile* и создается кнопка, основанная на получаемом изображении. Заметьте: *ImageFromFile* вызывается для каждого файла каталога! Если *ImageFromFile* завершается нормально, картинка загружена успешно. Если метод генерирует исключение, файл содержит изображение, не поддерживаемое методом, или файл поврежден.

Эту работу можно прервать до прочтения всего каталога. При вызове *ShowImages* все кнопки и подсказки очищаются, и процесс начинается заново для следующего каталога.

Кроме того, в классе *ImagePanel* определено открытое событие *ImageClicked*. Оно происходит при щелчке одной из кнопок. Два неизменяемых свойства *ClickedControl* и *ClickedImage* возвращают кнопку, которую щелкнули, и картинку на ней.

В программе ImageDirectory используются и *ImagePanel*, и *DirectoryTreeView*, а элемент управления *Splitter* разделяет их в клиентской области. Кроме того, в конструкторе создается объект *PictureBoxPlus* — класс, который я создал в главе 11. Он является усовершенствованием *PictureBox* и содержит свойство *NoDistort*, поддерживающее корректное сжатие при масштабировании изображения до размера элемента управления. Элемент управления *PictureBoxPlus* используется, чтобы растянуть изображение, которое щелкнули мышью, на всю клиентскую область. Первоначально свойство *Visible* этого объекта равно *false*.

ImageDirectory.cs

```
//---------------------------------------------
// ImageDirectory.cs © 2001 by Charles Petzold
//---------------------------------------------
using Petzold.ProgrammingWindowsWithCSharp;
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageDirectory: Form
{
     PictureBoxPlus    picbox;
     DirectoryTreeView dirtree;
     ImagePanel        imgpanel;
     Splitter          split;
     TreeNode          tnSelect;
     Control           cntlClicked;
     Point             ptPanelAutoScroll;

     public static void Main()
     {
          Application.Run(new ImageDirectory());
     }
     public ImageDirectory()
     {
          Text = "Image Directory";
          BackColor = SystemColors.Window;
```

см. след. стр.

```
        ForeColor = SystemColors.WindowText;

            // Создаем (невидимый) элемент управления для показа
            // большого изображений.

        picbox           = new PictureBoxPlus();
        picbox.Parent    - this;
        picbox.Visible   = false;
        picbox.Dock      - DockStyle.Fill;
        picbox.SizeMode  = PictureBoxSizeMode.StretchImage;
        picbox.NoDistort = true;
        picbox.MouseDown += new MouseEventHandler(PictureBoxOnMouseDown);

            // Создаем элементы управления для показа пиктограмм.

        imgpanel       = new ImagePanel();
        imgpanel.Parent = this;
        imgpanel.Dock   = DockStyle.Fill;
        imgpanel.ImageClicked +=
                    new EventHandler(ImagePanelOnImageClicked); .;

        split         = new Splitter();
        split.Parent    = this;
        split.Dock      = DockStyle.Left;
        split.BackColor - SystemColors.Control;

        dirtree        = new DirectoryTreeView();
        dirtree.Parent = this;
        dirtree.Dock   = DockStyle.Left;
        dirtree.AfterSelect +=
                  new TreeViewEventHandler(DirectoryTreeViewOnAfterSelect);

            // Создаем меню с одним пунктом (Refresh).

        Menu = new MainMenu();
        Menu.MenuItems.Add("&View");
        MenuItem mi - new MenuItem("&Refresh",
                          new EventHandler(MenuOnRefresh), Shortcut.F5);
        Menu.MenuItems[0].MenuItems.Add(mi);
    }
    void DirectoryTreeViewOnAfterSelect(object obj, TreeViewEventArgs tvea)
    {
        tnSelect = tvea.Node;
        imgpanel.ShowImages(tnSelect.FullPath);
    }
    void MenuOnRefresh(object obj, EventArgs ea)
    {
        dirtree.RefreshTree();
    }
    void ImagePanelOnImageClicked(object obj, EventArgs ea)
```

см.след.стр.

```
{
        // Получаем элемент управления и изображение,
        // которые щелкнули.

    cntlClicked = imgpanel.ClickedControl;
    picbox.Image - imgpanel.ClickedImage;

        // Сохраняем позицию автоматической прокрутки.

    ptPanelAutoScroll = imgpanel.AutoScrollPosition;
    ptPanelAutoScroll.X *= -1;
    ptPanelAutoScroll.Y *= -1;

        // Скрываем и отключаем обычные элементы управления.

    imgpanel.Visible = false;
    imgpanel.Enabled - false;
    imgpanel.AutoScrollPosition = Point.Empty;

    split.Visible в false;
    split.Enabled = false;

    dirtree.Visible = false;
    dirtree.Enabled = false;

        // Делаем видимым графическое окно.

    picbox.Visible = true;
}
    // Обработчики событий и методы, используемые при восстановлении
    // элементов управления

void PictureBoxOnMouseDown(object obj, MouseEventArgs mea)
{
    RestoreControls();
}
protected override void OnKeyDown(KeyEventArgs kea)
{
    if (kea.KeyCode == Keys.Escape)
        RestoreControls();
}
void RestoreControls()
{
    picbox.Visible - false;

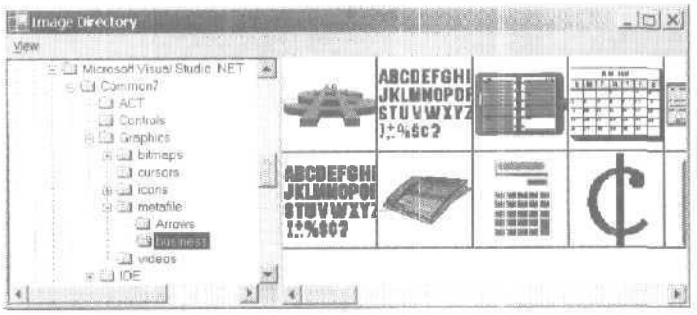
    dirtree.Visible = true;
    dirtree.Enabled = true;

    split.Enabled = true;
    split.Visible - true;
```

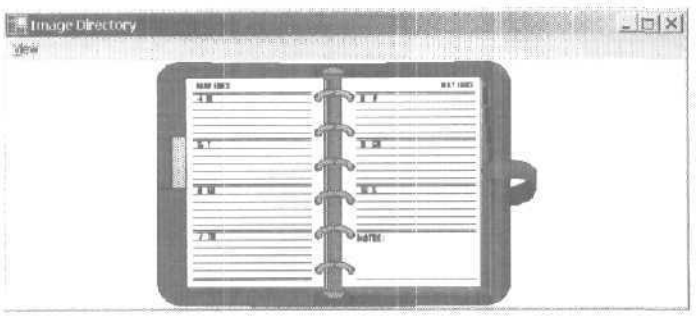
см. след. стр.

```
            imgpanel.AutoScrollPosition = ptPanelAutoScroll;
            imgpanel.Visible = true;
            imgpanel.Enabled = true;
            cntlClicked.Focus();
        }
    }
```

При изменении выбранного элемента в *DirectoryTreeView* (когда вызывается обработчик события *DirectoryTreeViewOnAfterSelect*) программа вызывает метод *ShowImages* класса *ImagePanel*. Вот как отображаются метафайлы одного из каталогов Visual Studio .NET:



При щелчке одной из кнопок программа оповещается об этом событии путем вызова обработчика события *ImagePanelOnImageClicked*. Обработчик делает три видимых элемента управления невидимыми, а невидимый элемент управления (*PictureBoxPlus*), наоборот, видимым. Изображение растягивается до размера клиентской области с соблюдением пропорций:



По щелчку в клиентской области или нажатию клавиши Esc клиентская область возвращается к обычному состоянию.

Основные сведения о списках

В самом сложном варианте элемент управления *ListView* показывает текстовую информацию в виде строк и столбцов, причем столбцы имеют заголовки. Первый столбец содержит *элементы* списка, остальные — *подэлементы,* связанные с каждым из элементов. Например, в Windows Explorer имя файла является элементом, а размер, дата изменения и атрибуты файла — подэлементами. Кроме того,

элемент управления *ListView* может показывать только простой список самих элементов (без подэлементов), элементы в несколько столбцов с мелкими значками и элементы в несколько столбцов с крупными значками.

При создании объекта *ListView* используется несколько различных классов. Начнем с самого класса *ListView*. У него несколько основных свойств:

Свойства *ListView* (выборочно)

Тип	Свойство	Доступ
View	*View*	Чтение/запись
ImageList	*SmallImageList*	Чтение/запись
ImageList	*LargeImageList*	Чтение/запись
List View:ColumnHeaderCollection	*Columns*	Чтение
ListView:ListViewItemCollection	*Items*	Чтение

Перечисление *View* содержит члены, соответствующие четырем форматам показа данных элементом управления *ListView*. Вам, наверное, знакомы эти форматы, так как вы их видели в разных меню, в частности, в панели инструментов стандартных диалоговых окон *OpenFileDialog* и *SaveFileDialog*:

Перечисление *View*

Член	Значение
Largelcon	О
Details	1
SmallIcon	*2*
List	3

Вариант *Largelcon* выводит элементы списка большими растровыми изображениями (48x48 пикселов) из свойства *LargelmageList*. Выбор других членов *View* выводит маленькие растровые изображения (16x16 пикселов) из свойства *SmallImageList*. Изображения из этих двух объектов *ImageList* должны соответствовать друг другу; например, третье изображение из *LargeImageList* должно быть увеличенной версией третьего изображения из *SmalllmageList*.

Свойство *Columns* — объект типа *ListView:ColumnHeaderCollection*, являющегося реализацией интерфейсов *ICollection*, *IEnumerable* и *IList*. Вот полный список его свойств:

Свойства *ListView.ColumnHeaderCollection*

Тип	Свойство	Доступ
ColumnHeader	*[]*	Чтение
in	*Count*	Чтение
boo!	*IsReadOnly*	Чтение

Как видите, объект типа *ListView:ColumnHeaderCollection* в основном является набором неизменяемых объектов *ColumnHeader*. В этом классе реализованы знакомые методы *Clear*, *Insert*, *Remove*, *Add* и *AddRange*. Вот как реализованы в этом классе последние два метода:

Методы *ListView.ColumnHeaderCollection* (выборочно)

```
int Add(ColumnHeader colhead)
ColumnHeader Add(string strText, int iWidth, HorizontalAlignment ha)
void AddRange(ColumnHeader[] acolheads)
```

По второй реализации метода *Add* можно сделать вывод, что объект *ColumnHeader* в основном состоит из текста, начальной ширины столбца в пикселах и выравнивания. Перечисление *HorizontalAlignment* уже встречалось вам раньше:

Перечисление *HorizontalAlignment*

Член	Значение
Left	**0**
Right	1
Center	2

Выравнивание — важный элемент заголовка столбца, так как влияет не только на текст в заголовке столбца, но и на элементы и подэлементы, перечисленные в этом столбце.

У самого *ColumnHeader* есть конструктор по умолчанию и всего три доступных для чтения и записи свойства, являющиеся тем же, что и параметры показанного выше метода *Add*:

Свойства *ColumnHeader* (выборочно)

Тип	Свойство	Доступ
string	*Text*	Чтение/запись
int	*Width*	Чтение/запись
HorizontalAlignment	*TextAlign*	Чтение/запись

Остальные свойства *ColumnHeader* — неизменяемые и показывают, к какому объекту *ListView* относится объект *ColumnHeader*, и индекс заголовка столбца в наборе заголовков столбцов,

Вернемся к таблице основных свойств *ListView,* Там в последней строке указано свойство *Items* — объект *ListView.ListViewItemCollection*.Вот свойства этого объекта:

Свойства *ListView.ListViewItemCollection*

Тип	Свойство	Доступ
ListViewItem	*0*	Чтение/запись
int	*Count*	Чтение/запись
bool	*IsReadOnly*	Чтение

Свойство *Items* класса *ListView* — это по сути набор объектов *ListViewItem*. Как обычно, намек на то, что из себя представляет *ListViewItem*, можно получить, посмотрев на методы *Add* и *AddRange* класса *ListView.ListViewItemCollection*:

Методы *ListView.ListViewItemCollection* (выборочно)

```
ListViewItem Add(ListViewItem lvitem)
ListViewItem Add(string strItem)
```

```
ListViewItem Add(string strItem, int indexImage)
void AddRange(ListViewItem[] alvitems)
```

Параметр *strItem* — это строка, относящаяся к элементу. Она показывается всегда независимо от выбранного варианта просмотра. При выборе *View.Details* также выводятся и подэлементы, которых здесь пока что не видно. Параметр *indexImage* является индексом как свойства *LargeImageList*, так и свойства *SmallImageList* элемента управления *ListView*.

У *ListViewItem* семь различных конструкторов:

Конструкторы *ListViewItem*

```
ListViewItem()
ListViewItem(string strItem)
ListViewItem(string strItem, int indexImage)
ListViewItem(string[] astrItems)
ListViewItem(string[] astrItems, int indexImage)
ListViewItem(string[] astrItems, int indexImage,
             Color clrFore, Color clrBack, Font font)
ListViewItem(ListViewItem.ListViewSubItem[] aSubItems, int indexImage)
```

При указании в конструкторе массива строк на самом деле указываются элемент и один или несколько подэлементов, связанных с ним,

Ключевую роль в классе *ListViewItem* играют свойства:

Свойства *ListViewItem* (выборочно)

Тип	Свойство	Доступ
string	*Text*	Чтение/запись
int	*ImageIndex*	Чтение/запись
object	*Tag*	Чтение/запись
ListViewItem.ListViewSubItemCollection	*SubItems*	Чтение

Объект *ListViewItem* содержит текст и индекс изображения, а также свойство *Tag*, позволяющее присоединять к элементу произвольные данные. Кроме того, *ListViewItem* содержит набор подэлементов — *ListViewItem.ListViewSubItemCollection*. Вот полный список свойств этого класса:

Свойства *ListViewItem.ListViewSubItemCollection*

Тип	Свойство	Доступ
ListViewItem.ListViewSubItem	*[]*	Чтение/запись
int	*Count*	Чтение
bool	*IsReadOnly*	Чтение

Как обычно, можно понять, что из себя представляет подэлемент, посмотрев на параметры методов *Add* и *AddRange* этого класса:

Методы *ListViewItem.ListViewSubItemCollection* (выборочно)

```
ListViewSubItem Add(string strText)
ListViewSubItem Add(string strText, Color clrFore, Color clrBack, Font font)
```

```
ListViewSubItem Add(ListViewItem.ListViewSubItem lvsi)
void AddRange(string[] astrText)
void AddRange(string[] astrText, Color clrFore, Color clrBack, Font font)
void AddRange(ListViewItem.ListViewSubItem[] alvsi)
```

Конструкторы *ListViewItem.ListViewSubItem* принимают аналогичные параметры:

Конструкторы *ListViewItem.ListViewSubItem*

```
ListViewItem.ListViewSubItem()
ListViewItem.ListViewSubItem(ListViewItem lviOwner, string strText)
ListViewItem.ListViewSubItem(ListViewItem lviOwner, string strText,
                             Color clrFore, Color clrText, Font font)
```

У этого класса только четыре свойства:

Свойства *ListViewItem.ListViewSubItem*

Тип	Свойство	Доступ
string	*Text*	Чтение/запись
Font	*Font*	Чтение/запись
Color	*BackColor*	Чтение/запись
Color	*ForeColor*	Чтение/запись

Пора применить всю эту информацию на практике. Надеюсь, вы не забыли серию программ SysInfo (информация о системе), которые находились в центре внимания главы 4 и показывались в некоторых последующих главах. Я хотел бы показать версию этой программы с элементом управления *ListView*. Кроме того, она использует класс *SysInfoReflectionStrings* обеспечивающий несколько открытых статических свойств и методов, наиболее важные из которых — свойства *Labels* и *Values*. Они возвращают массивы строк, показывающие соответственно имена статических свойств класса *SystemInformation* и их текущие значения. Свойство *Count* возвращает количество строк в массивах. Методы *MaxLabelWidth* и *MaxValueWidth* возвращают максимальную ширину строки в каждом из массивов. Я использовал эти методы в программе, чтобы присвоить начальные значения ширине столбцов.

SysInfoListView.cs

```
//-----------------------------------------
// SysInfoListView.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoListView: Form
{
    public static void Main()
    {
```

см. след. стр.

```
        Application. flun(new SysInfoListView());
    }
    public SysInfoListView()
    {
        Text = "System Information (List View)";

            // Создаем элемент управления ListView.

        ListView listview * new ListView();
        listview.Parent = this;
        listview.Dock = DockStyle.Fill;
        listview.View = View. Details;

            // Определяем столбцы, используя максимальную ширину строки.

        Graphics grfx = CreateGraphics();

        listview.Columns.Add("Property",
                 (int) SysInfoReflectionStrings.MaxLabelWidth(grfx, Font),
                 HorizontalAlignment.Left);

        listview.Columns.Add("Value",
                 (int) SysInfoReflectionStrings.MaxValueWidth(grfx, Font),
                 HorizontalAlignment.Left);

        grfx.Dispose();

            // Получаем показываемые данные.

        int      iNumItems  = SysInfoReflectionStrings.Count;
        string(3 astrLabels = SysInfoReflectionStrings.Labels;
        string[] astrValues = SysInfoReflectionStrings.Values;

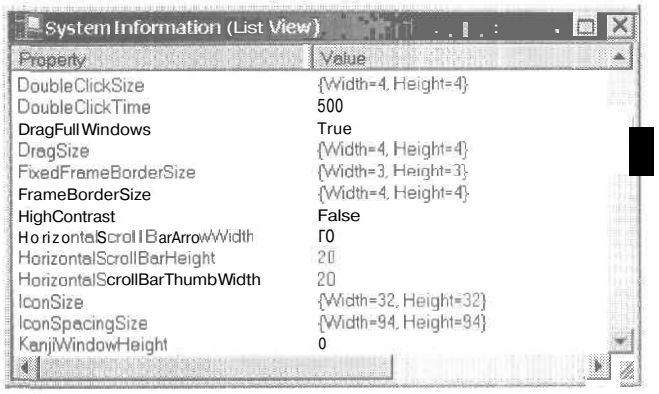
            // Определяем элементы и подэлементы.

        for (int i = 0; i < iNumItems; i++)
        {
            ListViewItem lvi = new ListViewItem(astrLabels[i]);
            lvi.SubItems.Add(astrValues[i]);
            listview.Items.Add(lvi);
        }
    }
}
```

Несмотря на длинные имена классов, используемые с наборами *ListView*. код оказался довольно коротким. Конструктор сначала создает объект *ListView,* делает форму предком этого объекта, присваивает свойству *Dock* объекта значение *DockStyle.Fill* и затем присваивает свойству *View* вариант отображения *View.Details.* (Другие варианты для этой программы не имели бы смысла.) Данный объект *ListView* не использует изображения,

Далее в конструкторе определяются два заголовка столбцов путем вызова использующего три параметра метода *Add* свойства *Columns*. Элементы и подэлементы добавляются в цикле *for* в конце. В нем программа создает объект *ListViewItem* для элемента массива *astrLabels*. Затем метод *Add* свойства *SubItems* добавляет один подэлемент массива *astrValues*. Затем объект *ListViewItem* добавляется к объекту *ListView* методом *Add* свойства *Items*.

Вот результат работы программы:



Полосы прокрутки присутствуют по умолчанию. Стоит поэкспериментировать с программой, чтобы посмотреть на возможности, по умолчанию отсутствующие в классе *ListView*, а также попробовать свойства, о которых я здесь не рассказывал.

События списков

Экспериментируя с программой SysInfoListView, вы обнаружите, что можете выбирать элементы первого столбца мышью или при помощи клавиш-стрелок «вверх» и «вниз». Удерживая клавишу Shift, можно распространить выделение на несколько элементов. Кроме того, можно выделять отдельные элементы и отменять их выделение (не затрагивая при этом выделение других элементов) щелчком при нажатой клавише Ctrl. (Чтобы отключить установленную по умолчанию возможность выделения нескольких элементов объекта *ListView*, присвойте свойству *MultiSelect* значение *false*.)

Программу может интересовать или не интересовать, изменил ли пользователь выделение. Но для большинства программ, использующих объект *ListView* для чего-нибудь большего, чем простой показ текста, представляет интерес так называемая *активизация* элемента. Так, Windows Explorer запускает приложение, когда пользователь активизирует элемент списка. По умолчанию активизация происходит при двойном щелчке элемента или группы элементов или нажатии клавиши Enter, когда выбран один или несколько элементов. Но это поведение позволяет изменить свойство:

Свойства *ListView (выборочно)*

Тип	Свойство	Доступ
ItemActivation	*Activation*	Чтение/запись

Перечисление *ItemActivation* содержит такие члены:

Перечисление *ItemActivation*

Член	Значение
Standard	0
OneClick	1
TwoClick	2

Standard — значение по умолчанию. При использовании *OneClick* и *TwoClick* элементы изменяют цвет при наведении на них курсора, для активизации с помощью *OneClick* требуется один щелчок, с помощью *TwoClick* — два щелчка.

Вот три самых важных события класса *ListView*:

События *ListView* (выборочно)

Событие	Метод	Делегат	Параметр
SelectedIndexChanged	*OnSelectedIndexChanged*	*EventHandler*	*EventArgs*
ItemActivate	*OnItemActivate*	*EventHandler*	*EventArgs*
ColumnClick	*OnColumnClick*	*ColumnClick-EventHandler*	*ColumnClick-EventArgs*

Ни один из других классов, связанных с элементом управления *ListView*, не содержит событий. Остальные события класса *ListView* связаны с редактированием элемента, установкой флажка для элемента и буксировкой элементов.

Кроме того, *ListView* поддерживает все события, реализованные в *Control*. Если, скажем, требуется определить контекстное меню в зависимости от элемента, который щелкнули правой кнопкой мыши, можно установить обработчик события *MouseDown* и определить, какой элемент щелкнули, методом *GetItemAt* класса *ListView*.

Щелчок заголовка столбца в событие *ColumnClick* дает следующую информацию:

Свойство *ColumnClickEventArgs*

Тип	Свойство	Доступ
int	*Column*	Чтение

Событиям *SelectedIndexChanged* и *ItemActivate* информация не сопутствует. При обработке этих событий для получения выделенных в этот момент элементов потребуются два свойства *ListView*.

Свойства *ListView* (выборочно)

Тип	Свойство	Доступ
ListView.SelectedIndexCollection	*SelectedIndices*	Чтение
ListView.SelectedListViewItemCollection	*SelectedItems*	Чтение

И, конечно, еще есть два набора! Первый — неизменяемый набор целых чисел:

Свойства *ListView.SelectedIndexCollection*

Тип	Свойство	Доступ
inl	*[]*	Чтение
int	*Count*	Чтение
bool	*IsReadOnly*	Чтение

Методы *Add* и *AddRange* этого класса не являются открытыми. Свойства второго набора:

Свойства *ListView.SelectedListViewItemCollection*

Тип	Свойство	Доступ
ListViewItem	*[]*	Чтение
int	*Count*	Чтение
bool	*IsReadOnly*	Чтение

И у этого набора методы *Add* и *AddRange* — неоткрытые. Для программного выделения элементов служит свойство *ListViewItem*:

Свойства *ListViewItem* (выборочно)

Тип	Свойство	Доступ
bool	*Selected*	Чтение/запись

Кроме того, это свойство позволяет получить выделенные элементы вместо *SelectedIndices* и *SelectedItems*. Нужно выполнить цикл по всем элементам *ListView* и проверить, свойство *Selected* каких из них имеет значение *true*.

Показанный ниже класс, *FileListView*, является наследником *ListView* и выводит список файлов, хранящихся в данном каталоге. В отличие от Windows Explorer *FileListView* не показывает подкаталоги вместе с файлами.

FileListView.cs

```
//-------------------------------------------
// FileListView.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Diagnostics;          // Для Process.Start
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class FileListView: ListView
{
     string strDirectory;

     public FileListView()
     {
          View = View.Details;

               // Получаем изображения для значков файлов.

          ImageList imglst = new ImageList();
          imglst.Images.Add(new Bitmap(GetType(), "DOC.BMP"));
          imglst.Images.Add(new Bitmap(GetType(), "EXE.BMP"));

          SmallImageList = imglst;
```

см. след. стр.

```
        LargeImageList = imglst;

            // Создаем столбцы.

        Columns.Add("Name",      100, HorizontalAlignment.Left);
        Columns.Add("Size",      100, HorizontalAlignment.Right);
        Columns.Add("Modified",  100, HorizontalAlignment.Left);
        Columns.Add("Attribute", 100, HorizontalAlignment.Left);
    }
    public void ShowFiles(string strDirectory)
    {
            // Сохраняем имя каталога как поле.

        this.strDirectory = strDirectory;

        Items.Clear();
        DirectoryInfo dirinfo = new DirectoryInfo(strDirectory);
        FileInfo[] afileinfo;

        try
        {
            afileinfo = dirinfo.GetFiles();
        }
        catch
        {
            return;
        }

        foreach (FileInfo fi in afileinfo)
        {
                // Создаем ListViewItem.

            ListViewItem lvi = new ListViewItem(fi.Name);

                // Присваиваем ImageIndex значение s зависимости
                // от расширения файла.

            if (Path.GetExtension(fi.Name).ToUpper() == ".EXE")
                lvi.ImageIndex = 1;
            else
                lvi.ImageIndex = 0;

                // Добавляем подэлементы и времени изменения размера
                // файла.

            lvi.SubItems.Add(fi.Length.ToString("NO"));
            lvi.SubItems.Add(fi.LastWriteTime.ToString());

                // Добавляем подэлемент атрибутов файла.
```

см. след. стр.

```
            string strAttr = "";

            if ((fi.Attributes & FileAttributes.Archive) != 0)
                strAttr += "A";

            if ((fi.Attributes & FileAttributes.Hidden) != 0)
                strAttr += "H";

            if ((fi.Attributes & FileAttributes.ReadOnly) != 0)
                strAttr += "R";

            if ((fi.Attributes & FileAttributes.System) != 0)
                strAttr += "S";

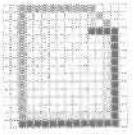
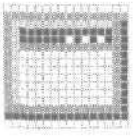
            lvi.SubItems.Add(strAttr);

                // Добавляем заполненный ListViewItem в FileListView.

            Items.Add(lvi);
        }
    }
    protected override void OnItemActivate(EventArgs ea)
    {
        base.OnItemActivate(ea);

        foreach (ListViewItem lvi in SelectedItems)
        {
            try
            {
                Process.Start(Path.Combine(strDirectory, lvi.Text));
            }
            catch
            {
                continue;
            }
        }
    }
}
```

Doc.bmp **Exe.bmp**

Чтобы получить значок для каждого показываемого файла, Windows Explorer, вероятно, использует функцию API *ExtractAssociatedIcon*. Однако для классов Windows Forms эта возможность недоступна. Чтобы все же показывать значки, FileListView загружает два растровых изображения, которые я скопировал из каталога

Program Files\Microsoft Visual Studio .NET\Common7\Graphics\Bitmaps\Outline\NoMask. Свойства *SmallImageList* и *LargeImageList* содержат одну и *my* же пару небольших картинок. В конструкторе создаются четыре столбца.

В классе *FileListView* определен открытый метод *ShowFiles*, выполняющий большую часть работы этого класса. Метод создает объект типа *DirectoryInfo* для указанного каталога и затем получает массив структур *FileInfo*, вызывая метод *GetFiles*. Каждому из членов массива ставится в соответствие элемент и три подэлемента. Большая часть кода связана с форматированием элементов. Если расширение файла — .exe, свойству *ImageIndex* присваивается 1, соответствующее изображению Exe.bmp; в противном случае ему присваивается 0, что соответствует изображению Doc.bmp. (Знаю, что вы спросите: «А как насчет файлов .dll? А насчет файлов .com?» Пожалуйста — усовершенствуйте программу так, чтобы она показывала значки, которые вы хотите.)

Кроме того, *FileListView* переопределяет метод *OnItemActivate*. Для каждого выбранного элемента программа вызывает статический метод *ProcessStart*. Если файл выполняемый, он запускается. Если это документ с известным ОС расширением, запускается приложение, связанное с документом.

Чтобы посмотреть, как выглядит элемент управления, нужна программа, аналогичная Windows Explorer, сочетающая *DirectoryTreeView* и *FileListView*. Вот она:

ExplorerUke.cs

```
//-----------------------------------------
// ExplorerLike.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ExplorerLike: Form
{
     FileListView      filelist;
     DirectoryTreeView dirtree;
     MenuItemView      mivChecked;

     public static void Main()
     {
          Application.Run(new ExplorerLikeO);
     }
     public ExplorerLike()
     {
          Text - "Windows Explorer-Like Program";
          BackColor - SystemColors.Window;
          ForeColor = SystemColors.WindowText;

               // Создаем элементы управления.

          filelist        = new FileListView();
          filelist.Parent = this;
```

см. след. стр.

```
        filelist.Dock   = DockStyle.Fill;

        Splitter split  = new Splitter();
        split.Parent    = this;
        split.Dock      = DockStyle.Left;
        split.BackColor * SystemColors.Control;

        dirtree         = new DirectoryTreeView();
        dirtree.Parent  - this;
        dirtree.Dock    = DockStyle.Left;
        dirtree.AfterSelect +=
                  newTreeViewEventHandler(DirectoryTreeViewOnAfterSelect);

            // Создаем меню View.

        Menu = new MainMenu();
        Menu.MenuItems.Add("&View"я);

        string[] astrView = { "Lar&ge Icons", "S&mall Icons",
                              "&List", "&Details" };
        View[] aview = { View.LargeIcon, View.SmallIcon,
                         View.List, View.Details };
        EventHandler eh = new EventHandler(MenuOnView);

        for (int i = 0; i < 4; i++)
        {
             MenuItemView miv = new MenuItemView();
             ffiiv.Text = astrView[i];
             miv.View = aview[i];
             miv.RadioCheck = true;
             miv.Click += eh;

             if (i == 3)          // По умолчанию -- View.Details
             {
                  mivChecked = mlv;
                  mivChecked.Checked = true;
                  filelist.View = mivChecked.View;
             }
             Menu.MenuItems[0].MenuItems.Add(miv);
        }
        Menu.MenuItems[0].MenuItems.Add("-");

            // Пункт меню View Refresh.

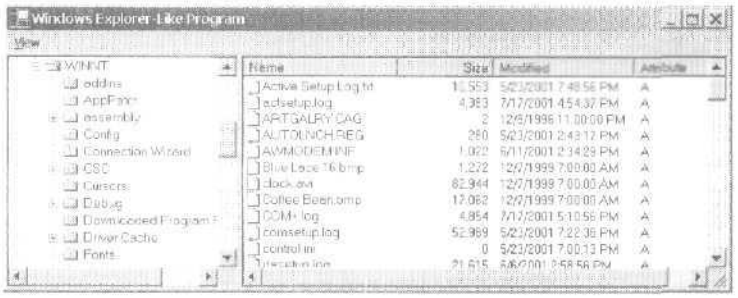
        MenuItem mi  = new MenuItem("&Refresh",
                           new EventHandler(MenuOnRefresh), Shortcut.F5);
        Menu.MenuItems[0].MenuItems.Add(mi);
    }
    void DirectoryTreeViewOnAfterSelect(object obj, TreeViewEventArgs tvea)
    I
```

см. след. стр.

```
            filelist.ShowFiles(tvea.Node.FullPath);
        }
        void MenuOnView(object obj, EventArgs ea)
        {
            fflivChecked.Checked = false;
            mivChecked = (MenuItemView) obj;
            mivChecked.Checked = true;

            filelist.View = mivChecked.View;
        }
        void MenuOnRefresh(object obj, EventArgs ea)
        {
            dirtree.RefreshTree();
        }
    }
    class MenuItemView: MenuItem
    {
        public View View;
    }
```

Большая часть программы связана с обработкой команд меню, позволяющих изменять свойство *View* элемента управления *FileListView* и обновлять содержимое *DirectoryTreeView*. Единственное связующее звено между этими двумя элементами — обработчик события *DirectoryTreeViewOnAfterSelect*, вызывающий метод *ShowFiles* класса *FileListView*, в который передается выбранный каталог. Вот как выглядит эта программа, показывающая часть системного каталога Windows:





ГЛАВА

23

Метафайлы

Метафайлы играют в векторной графике ту же роль, что битовые карты в растровой. Битовые карты обычно содержат реальные изображения, тогда как метафайлы обычно создает человек с помощью компьютерных программ. Метафайл состоит из последовательностей двоичных записей, соответствующих вызовам графических функций, рисующих прямые и кривые линии, закрашенные фигуры и текст. Кроме того, метафайлы могут содержать встроенные растровые изображения. Метафайлы могут храниться на жестком диске или целиком располагаться в памяти.

Растровые изображения создан IT в графических редакторах, метафайлы — в редакторах векторной графики. В хороших редакторах векторной графики можно перетаскивать объекты мышью, так как все компоненты рисунка хранятся как отдельные записи. В редакторах растровых изображений такой прием потребует серьезного анализа изображения. Возможности растровых редакторов обычно ограничены вставкой и перемещением прямоугольных участков изображения.

Поскольку метафайл описывает рисунок с помощью команд рисования графических объектов, такой рисунок можно масштабировать, не теряя разрешения, чего не скажешь о растровых изображениях. Если увеличить размер растрового изображения, его разрешение останется прежним, так как пикселы изображения просто копируются по вертикали и горизонтали. Любое сглаживание, применяемое при отображении масштабированных растровых изображений, убирает неровности, но при этом делает картинку размытой.

Метафайл можно преобразовать в растровое изображение, но при этом часть данных теряется: графические объекты, из которых состоял рисунок, объединяются в одно изображение и не могут изменяться по отдельности. Преобразование растровых изображений в метафайлы представляет собой гораздо более сложную задачу и обычно применяется только для очень простых изображений — ведь

для определения границ и контуров нужна большая вычислительная мощность. Однако, как я уже сказал, метафайлы могут содержать встроенные растровые изображения,

Сегодня метафайлы чаще всего применяются для передачи рисунков между программами через буфер обмена и для иллюстративных вставок. Поскольку метафайлы описывают рисунок в виде вызовов графических функций, они занимают меньше места и менее аппаратно зависимы, чем растровые изображения.

Однако визуализация метафайла длиться дольше, чем растрового изображения с таким же рисунком. Время вывода растрового изображения заданного размера и глубины цвета не зависит от сложности изображения, тогда как время вывода метафайла напрямую зависит от количества содержащихся в нем команд.

Не путайте метафайлы с графическими контурами! Контур — это просто набор координат, а метафайл также содержит описание используемых перьев и кистей. В графическом контуре текст сохраняется в виде очертаний символов, а в метафайле — в виде аргумента для вызова метода *DrawString*. Стандартного формата для сохранения контуров в файлах или передачи их через буфер обмена нет. тогда как метафайлы *разрабатывались* именно с целью сохранения в файлах и передачи через буфер обмена. (Об использовании метафайлов с буфером обмена я расскажу в главе 24.)

Метафайлы поддерживались в Windows, начиная с версии 1.0 (1985). Первоначальный формат метафайлов теперь называется Windows Metafile, а файлам, записанным в этом формате, присваивается расширение .wmf. С появлением 32-битных версий Windows метафайлы усовершенствованы, и введен новый формат Enhanced Metafile. Файлы, записанные в этом формате, имеют расширение .emf. 32-битные версии Windows поддерживают и старый и новый формат метафайлов.

Графическая система GDI+, используемая в Windows Forms, содержит ряд новых команд рисования, имеющих отношение к метафайлам. Усовершенствованные метафайлы, содержащие команды GDI+, называются метафайлами EMF+ (EMF-плюс), однако хранятся в EMF-файлах. Windows Forms также может создавать метафайлы, совместимые со стандартным форматом EMF, которые могут использоваться старыми 32-битными Windows-программами.

Загрузка и отображение метафайлов

Из главы 11 вы уже знаете, как загружать и отображать метафайлы. Для загрузки метафайла с диска можно использовать тот же статический метод *FromFile* класса *Image,* что и для растрового изображения:

```
Image image = Image.FromFile("PrettyPicture.emf");
```

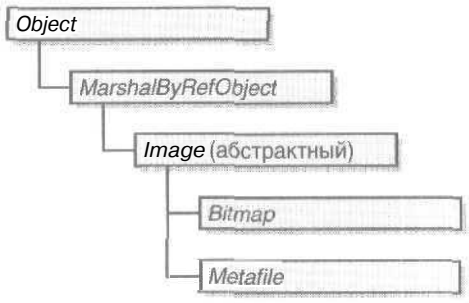
Так же осуществляется и вывод на экран. Для этого служит один из методов *DrawImage* или *DrawImageUnscaled* класса *Graphics*:

```
grfx.DrawImage(image, x, y);
```

Если на жестком диске есть WMF- или EMF-файлы (а если вы устанавливали хоть одно приложение, содержащее библиотеку иллюстраций, они наверняка там есть), то при помощи программы ImageIO из главы 16 эти метафайлы можно загрузить и вывести на экран. Набор метафайлов входит даже в комплект Visual Studio .NET.

По умолчанию они хранятся в папке C:\Program Files\Microsoft Visual Studio .NET\Common7\Graphics\Metafile.

Метафайлы считаются изображениями, поскольку — как и класс *Bitmap* — класс *Metafile* наследует классу *Image*:



Кроме того, как и *Bitmap,* класс *Metafile* является изолированным, т. с. не может наследоваться другими классами. Хотя классы *Image* и *Bitmap* определены в пространстве имен *System.Drawing,* класс *Metafile* и относящиеся к нему классы и перечисления определены в пространстве имен *System.Drawing.Imaging.* Если передать методу *GetType* значение, возвращаемое методом *Image.FromFile,* он вернет тип *System.Drawing.Bitmap* или *System.Drawing.Imaging.Metafile.*

При работе с метафайлами и объектами типа *Metafile* следует внимательно следить за терминологией. Метафайл — это набор инструкций по рисованию объектов, хранящийся на жестком диске или в памяти. Объект типа *Metafile*— это экземпляр класса *Metafile.* Статический метод *FromFile* класса *Image* создает объект типа *Metafile* на основе существующего метафайла,

Большую часть класса *Metafile* составляют 39 конструкторов, и несколько из них гораздо проще всех остальных. Создать объект *Metafile,* сославшись на существующий метафайл по имени файла или объекта типа *Stream,* можно, используя конструкторы:

Конструкторы *Metafile* (выборочно)

```
Metafile(string strFileName)
Metafile(Stream stream)
```

Эти два конструктора во многом эквивалентны соответствующим статическим методам *FromFile* класса *Image* за исключением того, что они явно возвращают объект типа *Metafile:*

```
Metafile mf = new Metafile("PrettyPicture.emf");
```

Так как класс *Metafile* наследуется от *Image,* для отображения метафайла используются такие же методы:

```
grfx.DrawImage(mf, x, y);
```

Разумеется, по отношению к метафайлу можно применять любые действия, поддерживаемые классом *Image.* Однако если вы загрузили метафайл из файла или потока, вам не удастся задействовать статический метод *FromImage* класса *Graphics,*

чтобы получить объект *Graphics* для изменения метафайла. Этот метод зарезервирован для новых метафайлов, создаваемых вашей программой.

Размеры и параметры отображения метафайлов

Как я говорил в главе 11, класс *Image* содержит свойства, описывающие изображение. Поскольку класс *Metafile* является потомком класса *Image,* эти свойства применимы и к метафайлам. При работе с метафайлами особенно полезны:

Свойства *Image* (выборочно)

Тип	Имя	Доступ
Size	*Size*	Чтение
int	*Width*	Чтение
int	*Height*	Чтение
float	*HorizontalResolution*	Чтение
float	*VerticalResolution*	Чтение
SizeF	*PhysicalDimension*	Чтение

Для объектов типа *Bitmap* свойства *Size, Width* и *Height* определяют размер изображения в пикселах, т. е. количество точек по вертикали и горизонтали. Свойства *HonzontalResolution* и *VerticalResolution* сообщают разрешение, выраженное в количестве пикселов на дюйм по вертикали и горизонтали. Эти данные могут содержаться в растровом файле. Разделив значение свойства *Width* на значение *HonzontalResolution,* а свойства *Height* — на значение *VerticalResolution,* можно получить размер изображения в дюймах. Если затем преобразовать полученные значения в миллиметры и умножить на 100, получится значение свойства *PhysicalDimension,* содержащее размер картинки в сотых долях миллиметра.

Для объектов типа *Metafile* свойства *Size, Width* и *Height* имеют несколько иное значение. Часто они отражают координаты и размеры всех объектов метафайла. Так, если метафайл состоит из единственного вызова метода *DrawLine* с координатами концов отрезка, равными (-50, 25) и (100, 250), значение *Width,* вероятно, будет равно 150 (или около того), а *Height* — около 225. Однако, как мы вскоре увидим, создатель метафайла может установить иные значения *Width* и *Height.* Кроме того, если в рисунке применяются широкие линии, они тоже могут повлиять на размер изображения, а следовательно, и на значения *Width* и *Height.* Так что, хотя метафайлы не содержат пикселов, они характеризуются чем-то вроде размера в пикселах.

Кроме того, объекты типа *Metafile* характеризуются свойствами *HorizontalResolution* и *VerticalResolution,* показывающими отношение координат метафайла к дюймам. Значения свойств *HorizontalResolution* и *VerticalResolution* гипотетического метафайла с единственным вызовом *DrawLine,* о котором мы говорили чуть выше, могли равняться 75, что соответствовало бы изображению шириной в 2 и высотой в 3 дюйма. Свойство *PhysicalDimension* в таком случае могло содержать значение (5080, 7620).

Отобразить метафайл в соответствии с его метрическими размерами, расположив верхний левый угол в координатах (x, y), позволяет оператор:

```
DrawImage(mf, x, y);
```

или один из его вариантов, которому передаются аргументы типа *Point* или *PointF*. Отображаемый размер не изменяется преобразованиями страницы, но на него влияют глобальные преобразования.

Следующий метод *DrawImage* (и его варианты, использующие аргументы *Rectangle* и *RectangleF*) отображает метафайл, вписанным в заданный прямоугольник:

```
DrawImage(mf, x, y, cx, cy);
```

На интерпретацию аргументов *x, y, cx и cy* влияют и преобразования страницы, и глобальные преобразования. Чтобы отобразить метафайл в оригинальном размере, установите пикселы в качестве единиц измерения для страницы и примените оператор:

```
DrawImage(mf, x, y, mf.Width, mf.Height);
```

Класс *Metafile* не содержит дополнительных открытых свойств, кроме тех, что он наследует от класса *Image*. Однако в самом метафайле существует заголовок, содержащий дополнительные сведения о файле. Этот заголовок инкапсулируется в классе *MetafileHeader*. Получить экземпляр этого класса позволяет:

Нестатический метод *GetMetafileHeader* класса *Metafile*

```
MetafileHeader GetMetafileHeader()
```

Если вы не создавали объект типа *Metafile,* получить заголовок метафайла можно, используя один из статических методов:

Статические методы *GetMetafileHeader* класса *Metafile* (выборочно)

```
MetafileHeader GetMetafileHeader(string strFileName)
MetafileHeader GetMetafileHeader(Stream stream)
```

Есть еще два статических метода *GetMetafileHeader* для работы с описателями Win32,

Класс *MetafileHeader* содержит 10 неизменяемых свойств. Вот пять из них:

Свойства *MetafileHeader* (выборочно)

Тип	Имя	Доступ
MetafileType	*Type*	Чтение
int	*Version*	Чтение
int	*MetafileSize*	Чтение
int	*EmfPlusHeaderSize*	Чтение
MetaHeader	*WmfHeader*	Чтение

Свойство *Type* указывает тип метафайла и может принимать одно из значений перечисления *MetafileType*:

Перечисление *MetafileType*

Член	Значение
Invalid	0
Wmf	1
WmfPlaceable	2

Перечисление *MetafileType* *(продолжение)*

Член	Значение
Emf	3
EmfPlusOnly	4
EmfPlusDual	5

Члены с префиксом *Wmf* соответствуют старому 16-битному формату метафайлов. Метафайлы, обозначенные членами с префиксом *Emf*, создаются в 32-битных Windows-приложениях с использованием Windows API или MFC. Как мы увидим, метафайлам, созданным Windows Forms, по умолчанию присваивается тип *EmfPlusDual*, означающий, что эти файлы содержат как GDI-, так и GDI+-записи. Такие метафайлы совместимы с Win32-программами. Метафайлы типа *EmfPlusOnly* содержат только записи GDI+ и могут использоваться только программами Windows Forms.

Свойство *MetafileSize* указывает размер метафайла. Для метафайлов, хранящихся на жестком диске, его значение равно размеру файла. Свойство *WmfHeader* содержит дополнительные сведения о метафайлах типа WME

Ниже приведены все методы класса *MetafileHeader*, предоставляющие булев интерфейс для свойства *Type*:

Методы *MetafileHeader*

```
bool IsWmf()
bool IsWmfPlaceable()
bool IsEmf()
bool IsEmfPlus()
bool IsEmfPlusOnly()
bool IsEmfPlusDual()
bool IsEmfOrEmfPlus()
bool IsDisplay()
```

Для метафайлов типа *MetafileType.Emf* методы *IsEmf* и *IsEmfOrEmfPlus* возвращают значение *true*. Методы *IsEmfPlus*, *IsEmfPlusOnly* и *IsEmfOrEmfPlus* возвращают *true* для метафайлов типа *MetafileType.EmfPlusOnly*. А для метафайлов *MetafileType.EmfPlusDual* значение *true* возвращают методы *IsEmfPlus*, *IsEmfPlusDual* и *IsEmfOrEmfPlus*.

Как мы увидим, метафайл всегда создается для определенного устройства графического вывода. Метод *IsDisplay* возвращает *true* для метафайлов, созданных для видеодисплея, и *false* — для принтера.

Вот оставшиеся свойства класса *MetafileHeader*:

Свойства *MetafileHeader* (выборочно)

Тип	Имя	Доступ
Rectangle	*Bounds*	Чтение
float	*DpiX*	Чтение
float	*DpiY*	Чтение
int	*LogicalDpiX*	Чтение
int	*LogicalDpiY*	Чтение

Значения полей *Width* и *Height* свойства *Bounds* должны соответствовать значениям свойств *Width* и *Height,* унаследованных объектом типа *Metafile* от класса *Image.* Значения *DpiX* и *DpiY* должны соответствовать значениям свойств *HorizontalResolution* и *VerticalRsolution* класса *Image.* Свойства *LogicalDpiX* и *LogicalDpiY* в программах Windows Forms не используются.

Поля *X* и *Y* свойства *Bounds* не обязательно должны иметь значение 0. Так, выше я говорил о гипотетическом метафайле, состоящем из единственного вызова *DrawLine* с координатами (–50, 25) и (100, 250). Свойство *Bounds* заголовка метафайла обычно определяет наименьший прямоугольник, куда вписываются все объекты метафайла. Таким образом, *Bounds* должно определять прямоугольник с координатами (-50. 25, 150, 225).

На самом деле в этом случае *Bounds* скорее всего будет содержать координаты (–51, 24, 153, 228). Поскольку в системе GDI+ линии чертятся до второй точки и она включается в длину линии, то общая длина линии на 1 пиксел превышает вычисленную. Кроме того, линии имеют конечную ширину, увеличивающую размеры еще на 1 пиксел с каждого конца. И, наконец, сама программа, создающая метафайл, может устанавливать значение свойства *Bounds* отличным от подразумеваемого содержимым метафайла.

Начальная точка прямоугольника, определяемого свойством *Bounds,* т е. значениями его полей *Xu Y,* не влияет на положение рисунка при визуализации. Так, если отобразить рассмотренный ранее гипотетический метафайл оператором:

```
grfx.DrawImage(mf, 0, 0);
```

вы сможете увидеть его целиком. Верхний левый угол прямоугольника *Bounds* отображается в точке, указанной при вызове метода *DrawImage.* В приведенном примере это точка с координатами (О, О).

Следующая программа содержит объект *OpenFileDialog* для загрузки и отображения содержимого метафайла.

MetafileViewer.cs

```
//-----------------------------------------
// MetafileViewer.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Drawing.Printing;
using System.IO;              // Для класса Path
using System.Windows.Forms;

class MetafileViewer: Form
{
     protected Metafile mf;
     protected string   strProgName;
     protected string   strFileName;
     MenuItem           miFileSaveAs, rolFilePrint,
                        miFileProps, miViewChecked;
```

см. след. стр.

```
public static void Main()
{
     Application.Run(new MetafileViewer());
}
public MetafileViewer()
{
     Text = strProgName + "Metafile Viewer";
     ResizeRedraw = true;

     Menu = new MainMenu();

          // Меню File

     MenuItem ml = new MenuItem("&File");
     mi.Popup += new EventHandler(MenuFileOnPopup);
     Menu.MenuItems.Add(mi);

          // Пункт Open из меню File

     mi = new MenuItem("&Open...");
     ffli.Click += new EventHandler(MenuFileOpenOnClick);
     mi.Shortcut = Shortcut.CtrlO;
     Menu.MenuItems[0].MenuItems.Add(mi);

          // Пункт Save As Bitmap из меню File

     miFileSaveAs = new MenuItem("Save &As Bitmap...");
     miFileSaveAs.Click += new EventHandler(MenuFileSaveAsOnClick);
     Menu.MenuItems[0].MenuItems.Add(miFileSaveAs);
     Menu.MenuItems[0].MenuItems.Add("-");

          // Пункт Print из меню File

     miFilePrint = new MenuItem("&Print...");
     miFilePrint.Click += new EventHandler(MenuFilePrintOnClick);
     Menu.MenuItems[0].MenuItems.Add(miFilePrint);
     Menu.MenuItems[0].MenuItems.Add("-");

          // Пункт Properties из меню File

     miFileProps = new MenuItem("Propert&ies...");
     miFileProps.Click += new EventHandler(MenuFilePropsOnClick);
     Menu.MenuItems[0].MenuItems.Add(miFileProps);

          // Меню Edit (для использования в главе 24)

     Menu.MenuItems.Add("&Edit");

          // Меню View
```

си. *след. стр.*

```
        Menu.MenuItems.Add("&View");

        string[] astr = { "&Stretched to Window",
                          "&Metrical Size", "&Pixel Size" };
        EventHandler eh = new EventHandler(MenuViewOnClick);

        foreach (string str in astr)
            Menu.MenuItems[2].MenuItems.Add(str, eh);

        miViewChecked = Menu.MenuItems[2].MenuItems[0];
        miViewChecked.Checked = true;
    }
    void MenuFileOnPopup(object obj, EventArgs ea)
    {
        miFileSaveAs.Enabled = ;
        miFilePrint.Enabled =
        miFileProps.Enabled = (af != null);
    }
    void MenuFileOpenOnClick(object obj, EventArgs ea)
    {
        OpenFileDialog dlg = new OpenFileDialog();

        dlg.Filter = "All Metafiles|*.wmf;*.emf|" +
                     "Windows Metafile (*.wmf)|*.wmf|" *
                     "Enhanced Metafile (*.emf)|*.emf|" +
                     "All files|*.*";

        if (dlg.ShowDialog() == DialogResult.OK)
        {
            try
            {
                mf = new Metafile(dlg.FileName);
            }
            catch (Exception exc)
            {
                MessageBox.Show(exc.Message, strProgName);
                return;
            }
            strFileName = dlg.FileName;
            Text = strProgName + " - " + Path.GetFileName(strFileName);
            Invalidate();
        }
    }
    protected virtual void MenuFileSaveAsOnClick(object obj, EventArgs ea)
    {
        MessageBox.Show("Not yet implemented!", strProgName);
    }
    void MenuFilePrintOnClick(object obj, EventArgs ea)
    {
        PrintDocument prndoc = new PrintDocument();
```

см. след. стр.

```
        prndoc.DocumentName = Text;
        prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
        prndoc.Print();
    }
    void MenuFilePropsOnClick(object obj, EventArgs ea)
    {
        MetafileHeader rah = rcf.QetMetafileHeaderO;

        string str =
            "Image Properties" +
            "\n\tSize * " + mf.Size *
            "\n\tHorizontal Resolution = " + mf.HorizontalResolution +
            "\n\tVertical Resolution = " + mf.VerticalResolution +
            "\n\tPhysical Dimension = " + mf.PhysicalDimension +
            "\n\nMetafile Header Properties" +
            "\n\tBounds o " + mh.Bounds +
            "\n\tDpiX = " + rah.DpiX +
            "\n\tDpiY = " + rah.DpiY +
            "\n\tLogicalDpiX = " + mh.LogicalDpiX +
            "\n\tLogicalDpiY = " + mh.LogicalDpiY +
            "\n\tType = " + fflh.Type +
            "\n\tVersion - " + mh.Version *
            "\n\tMetafileSize * " + mh.MetafileSize;

        MessageBox.Show(str, Text);
    }
    void MenuViewOnClick(object obj, EventArgs ea)
    {
        miViewChecked.Checked = false;
        miViewChecked - (MenuItem) obj;
        miViewChecked.Checked - true;
        Invalidate();
    Г
    void OnPrintPage(object obj, PrintPageEventArgs ppea)
    {
        Graphics  grfx = ppea.Graphics;
        Rectangle rect = new Rectangle(
            ppea.MarginBounds.Left -
            (ppea.PageBounds.Width -
                (int) grfx.VisibleClipBounds.Width) / 2,
            ppea.MarginBounds.Top -
            (ppea.PageBounds.Height -
                (int) grfx.VisibleClipBounds.Height) / 2,
            ppea.MarginBounds.Width,
            ppea.MarginBounds.Height);

        DisplayMetafile(grfx, rect);
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
```

см.след.стр.

```
        if (mf != null)
            DisplayMetafile(pea.Graphics, ClientRectangle);
    }
    void DisplayMetafile(Graphics grfx, Rectangle rect)
    {
        switch (miViewChecked.Index)
        {
        case 0:  grfx.DrawImage(mf, rect);   break;
        case 1:  grfx.DrawImage(mf, rect.X, rect.Y);   break;
        case 2: flrfx.DrawlifiaoeCraf, rect.X, rect.Y, mf.Width, mf.Height);
                 break;
        }
    }
}
```

Эта программа несколько отличается от ImageOpen из главы 16 (которая тоже способна загружать и отображать метафайлы). Во-первых, в ее меню File включен пункт Properties, отображающий важнейшие сведения о метафайле на основе значений *Image* и *MetafileHeader*.Кроме того, она содержит три пункта меню, управляющие способом отображения метафайла при помощи метода *DrawImage*. Пункт Print позволяет распечатать рисунок с учетом выбранного способа отображения.

На данном этапе действия для некоторых пунктов меню остались нереализованными. Меню Edit будет целиком описано в главе 24, а пункт Save As Bitmap из меню File мы задействуем в следующем разделе,

Преобразование метафайлов в растровые изображения

Я уже говорил, что преобразовать метафайл в растровое изображение довольно просто. Вообще-то механизм преобразования встроен в Windows Forms. Загрузив метафайл при помощи программы ImageIO, вы сохраните его в виде растрового изображения.

Но, возможно, вы захотите реализовать такое преобразование самостоятельно. Это может потребоваться, если вам нужен полный контроль над процессом или если вы не хотите записывать растровое изображение на диск. Или, быть может, вы имеете дело с метафайлом, содержащим большое количество команд, и преобразование его в растровое изображение сможет ускорить вывод на экран.

Следующий потомок MetafileViewer добавляет метод *MetafileToBitmap* преобразующий объект типа *Metafile* в объект типа *Bitmap*. После преобразования программа записывает полученное растровое изображение на диск (как это делала и ImageIO), но реализованный в ней метод можно использовать и в других целях.

```
Bitmap bm = new Bitmap(mf);
```

Полученное таким образом растровое изображение будет иметь тот же размер в пикселах, что и метафайл, но его разрешение будет равно разрешению экрана.

Создание новых метафайлов

Пока мы рассмотрели только 2 из 39 конструкторов *класса Metafile*. Это конструкторы, которые загружают существующий метафайл, определяемый именем файла или объектом типа *Stream*. Три конструктора *класса Metafile* создают объект для метафайла, применяя описатели Win32, и могут пригодиться при сопряжении с уже написанными программами. Прочие 34 конструктора создают новый метафайл, что очень часто подразумевает создание нового файла на диске или удаление содержимого существующего файла при подготовке к созданию нового метафайла. Вот простейшие конструкторы для создания нового метафайла:

Конструкторы *Metafile* (выборочно)

```
Metafile(string strFileName, IntPtr ipHdc)
Metafile(Stream stream, IntPtr ipHdc)
```

Второй аргумент этих конструкторов довольно странен. Чтобы получить информацию о разрешении, метафайлу нужно сопоставить устройство для вывода графики, но для этого логичнее было бы применить в качестве второго аргумента конструкторов объект типа *Graphics*:

```
Metafile mf = new Metafile("NewFile.emf", grfx);    // Увы, неверно!
```

Такая реализация совпала бы с конструктором класса *Bitmap* в описанном мною методе *MetafileToBitmap*.

Вместо этого вторым аргументом конструктору класса *Metafile* передается описатель контекста устройства Win32. Объект *Graphics* инкапсулирует контекст устройства Win32, так что для получения и освобождения описателя потребуются методы *GetHdc* и *ReleaseHdc* класса *Graphics*:

```
IntPtr ipHdc = grfx.GetDC();
Metafile mf = new Metafile("NewFile.emf", ipHdc);
grfx.ReleaseDC(ipHdc);
```

Обычно метафайлы создаются на основе разрешения экрана, но не стоит создавать его в методе *OnPaint*. Как вы знаете, в любом классе — потомке *Control* объект *Graphics* можно получить методом *CreateGraphics*. Для использованного объекта *Graphics* следует вызвать метод *Dispose*. Таким образом, создание нового метафайла будет выглядеть примерно так:

```
Graphics grfxVideo = CreateGraphics();
IntPtr ipHdc = grfxVideo.GetDC();
Metafile mf = new MetafileC'NewFile.emf", ipHdc);
grfxVideo.ReleaseDC(ipHdc);
grfxVideo.Dispose();
```

Конечно этот код длиннее эквивалентного кода Win32. Один из вариантов функции Win32 для создания метафайлов позволяет при указании дисплея задейство-

вать устройство *null,* но, к сожалению, его нельзя использовать в конструкторе класса *Metafile.*

После создания метафайла вам понадобится еще один экземпляр объекта *Graphics,* чтобы внести в метафайл команды для рисования графики. Этот экземпляр можно получить тем же методом, что и при рисовании растрового изображения:

```
Graphics grfxMetafile = Graphics.FromImage(mf);
```

Для большей ясности я присвоил этому объекту *Graphics* другое имя. Но так как в программе данные объекты не пересекаются, создать их может одна переменная,

Объект *Graphics,* полученный методом *FromImage,* можно передавать любому методу рисования графики, а в метафайл вносится запись для каждого вызванного метода. Кроме того, класс *Graphics* содержит метод, используемый только для метафайлов:

Методы *Graphics* (выборочно)

void AddMetafileComment(byte[] abyData)

Создав рисунок, надо вызвать метод *Dispose* для второго объекта *Graphics:*

```
grfxMetafile.Dispose();
```

Эта небольшая программа, наследующая класс *PrintableForm,* создаст метафайл в своем конструкторе; в методе *DoPage* она получает размер метафайла и заполняет рисунком клиентскую область или страницу при печати:

CreateMetafile.cs

```
//---------------------------------------------
// CreateMetafile.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;              // Пока не используется!
using System.Windows.Forms;

class CreateMetafile: PrintableForm
{
     Metafile mf;

     public new static void Main()
     {
          Application.Run(new CreateMetafile());
     }
     public CreateMetafile()
     {
          Text = "Create Metafile";

               // Создание нового метафайла.
```

см. след. стр.

```
        Graphics grfx = CreateGraphics();
        IntPtr ipHdc = grfx.GetHdc();

        mf = new Metafile("CreateMetafile.emf", ipHdc);

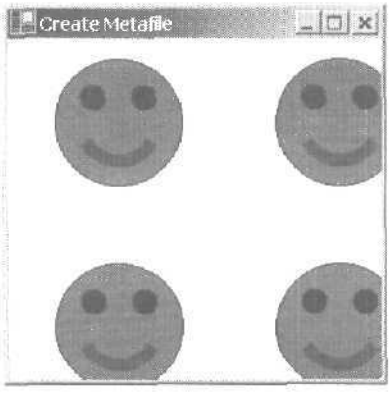
        grfx.ReleaseHdc(ipHdc);
        grfx.Dispose();

            // Создание рисунка в метафайле.

        grfx = Graphics.FromImage(mf);

        grfx.FillEllipse(Brushes.Gray, 0, 0, 100, 100);
        grfx.DrawEllipse(Pens.Black, 0, 0, 100, 100);
        grfx.FillEllipse(Brushes.Blue, 20, 20, 20, 20);
        grfx.FillEllipse(Brushes.Blue, 60, 20, 20, 20);
        grfx.DrawArc(new Pen(Color.Red, 10), 20, 20, 60, 60, 30, 120);
        grfx.Dispose();
    }
    protected override void DoPageCGraphics grfx, Color clr, int cx, int ey)
    {
        for (int y = 0; y < cy; y += mf.Height)
        for (int x = 0; x < cx; x += mf.Width)
            grfx.DrawImage(mf, x, y, mf.Width, mf.Height);
    }
}
```

Рисунок, помещаемый в метафайл, содержит вызовы методов *DrawEllipse*, *FiilEllipse* и *DrawArc*. Результат работы этих методов — с использованием соответствующих перьев и кистей — выводится в клиентской области методом *DoPage*. Отобразить рисунок позволяет метод *DrawImage*, которому в качестве аргументов передается ширина и высота рисунка в пикселах:



Обратить внимание надо только на то, что картинки не прилегают друг к другу. Граница вокруг каждого изображения является побочным эффектом технологии, используемой классом *Metafile* для преобразования координат с плавающей точ-

кой GDI+ в целочисленные координаты GDI в записях метафайла. Вскоре мы увидим, как управлять размерами границы, чтобы убрать излишки по краям рисунка.

Для отображения рисунка согласно его размеру в пикселах (а не метрическому размеру) в программе CreateMetafile применяется такая форма метода *DrawImage*:

```
grfx.DrawImage(mf, x, y, mf.Width, mf.Height);
```

Если бы вывод осуществлялся только на экран, я мог бы использовать и метрические размеры:

```
grfx.DrawImage(mf, x, y);
```

Поскольку разрешение метафайла совпадает с разрешением экрана, результаты оказались бы одинаковыми. Однако рисунки, распечатанные на принтере при помощи этих двух вызовов метода *Drawlmage,* скорее всего получились бы разного размера. Как и в случае с растровыми изображениями, метафайлы удобнее располагать на экране или листе, если выразить размер рисунка в пикселах.

В качестве альтернативы можно заменить тело метода *DoPage* оператором:

```
grfx.FillRectangle(new TextureBrush(mf), 0, 0, cx, cy);
```

который создает из метафайла объект *TextureBrush* и заполняет им клиентскую область. А вы думали, что текстурные кисти создаются только из растровых изображений!

При каждом запуске CreateMetafile заново создает файл CreateMetafile.emf, но делать это совсем не обязательно. В конструктор *CreateMetafile* сразу за установкой свойства *Text* можно попробовать добавить такой код:

```
if (File.Exists("CreateMetafile.emf"))
{
     mf = new Metafile("CreateMetafile.emf");
     return;
}
```

Если файл существует, этот участок программы загружает его и выполняет выход из конструктора. (Статический метод *Exists* класса *File* определен в пространстве имен *SystemIO*. Программа CreateMetafile уже содержит оператор *using* для данного пространства, хотя оно требуется только для этой проверки.)

CreateMetafile сохраняет объект *Metafile* (с именем *mf)* как поле. Это нужно, чтобы созданный в конструкторе метафайл мог использоваться в методе *DoPage*. Однако необязательно сохранять объект *Metafile*. Метод *DoPage* может загрузить метафайл сам. Программа CreateMetafileReload аналогична CreateMetafile: создает метафайл, только если он еще не создан, но объект *Metafile* не сохраняется в поле. Вместо этого в программе сохраняется только имя файла в виде строки, а метод *DoPage* сам загружает метафайл.

CreateMetafileReload.cs

```
//-----------------------------------------------
// CreateMetafileReload.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
```

см. след. стр.

```
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;

class CreateMetafileReload: PrintableForm
{
    const string strMetafile = "CreateMetafileReload.emf";

    public new static void Main()
    {
        Application.Run(new CreateMetafileReload());
    }
    public CreateMetafileReloadO
    {
        Text = "Create Metafile (Reload)";

        if (!File.Exists(strMetafile))
        {
                // Создание нового метафайла.

            Graphics grfx = CreateGraphics();
            IntPtr ipHdc = grfx.GetHdc();

            Metafile mf - new Metafile(strMetafile, ipHdc);

            grfx.ReleaseHdc(ipHdc);
            grfx.Dispose();

                // Создание рисунка в метафайле.

            grfx = Graphics.FromImage(mf);

            grfx.FillEllipse(Brushes.Gray, 0, 0, 100, 100);
            grfx.DrawEllipse(Pens.Black, 0, 0, 100, 100);
            grfx.FillEllipse(Brushes.Blue, 20, 20, 20, 28);
            grfx.FillEllipse(Brushes.Blue, 60, 20, 29, 20);
            grfx.DrawArc(new Pen(Color.Red, 10), 20, 20, 60, 60, 30, 120);
            grfx.Dispose();
        }
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Metafile mf = new Metafile(strMetafile);

        for tint y = 0; y < cy; y += mf.Height)
        for tint x = 0; x < cx; x += mf.Width)
            grfx.DrawImage(mf, x, y, mf.Width, mf.Height);
    }
}
```

Я приводил примеры применения конструкторов класса *Metafile*, использующих в качестве аргумента строку с именем файла. Вы также вправе задействовать аргумент типа *Stream*. Так, если заменить конструктор метода *DoPage* следующим кодом, он будет работать точно так же:

```
FileStream fs = new FileStream(strMetafile, FileMode.Open);
Metafile mf = new Metafile(fs);
fs.Closed
```

Но он *не* будет работать, если конструктор *FileStream* будет открывать файл только для записи или если аргумент *FileMode* будет задавать уничтожение содержимого файла.

Для создания объекта *Metafile* в конструкторе программы можно использовать объект *FileStream*:

```
FileStream fs = new FileStream(strMetafile, FileMode.Create);
Metafile mf = new Metafile(fs, ipHdc);
```

Заметьте: аргумент *FileMode* указывает на то, что файл должен быть создан заново. После всех вызовов методов объекта *Graphics* поток надо закрыть:

```
fs.Close();
```

Кроме того, *MemoryStream* позволяет создать метафайл в оперативной памяти. Поскольку объекты *MemoryStream* не имеют имен, сам объект *MemoryStream* (или объект *Metafile*) должен сохраняться в поле.

Вернемся к программе CreateMetafile. Если вы последовали моему совету и вставили в нее фрагмент кода с вызовом метода *File£xists*, удалите его. Теперь замените конструктор объекта *Metafile*:

```
mf = new Metafile("CreateMetafile.emf", ipHdc);
```

на:

```
mf = new Metafile(new MemoryStream(), ipHdc);
```

Класс *MemoryStream* определен в пространстве имен *SystemIO*. Программа с таким конструктором создает и использует метафайл в оперативной памяти. При этом файл на жестком диске не создается.

В этой программе объект *MemoryStream* (но не *Metafile*) сохраняется в виде поля:

CreateMetafileMemory.cs

```
//-------------------------------------------------
// CreateMetafileMemory.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;

class CreateMetafileMemory: PrintableForm
```

см. след. стр.

```
{
    readonly MemoryStream ms = new MemoryStream();

    public new static void Main()
    {
        Application.Run(new CreateMetafileMemory());
    }
    public CreateMetafileMemory()
    {
        Text = "Create Metafile (Memory)";

            // Создание нового метафайла.

        Graphics flrfx = CreateGraphics();
        IntPtr ipHdc = grfx.GetHdc();

        Metafile mf = new Metafile(ms, ipHdc);

        grfx.ReleaseHdc(ipHdc);
        grfx.Dispose();

            // Создание рисунка в метафайле.

        grfx = Graphics.FromImage(mf);

        grfx.FillEllipse(Brushes.Gray, 0, 0, 100, 100);
        grfx.DrawEllipse(Pens.Black, 0, 0, 100, 100);
        grfx.FillEllipse(Brushes.Blue,20,20,20,20);
        grfx.FillEllipse(Brushes.Blue, 60, 20, 20, 20);
        grfx.DrawArc(new Pen(Color.Red, 10), 20, 20, 60, 60, 30, 120);
        grfx.Dispose();
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        ms.Position = 0;
        Metafile mf = new Metafile(ms);

        for (int y = 0; y < cy; y += mf.Height)
        for (int x = 0; x < cx; x += mf.Width)
            grfx.DrawImage(mf, x, y, mf.Width, mf.Height);
    }
}
```

Заметьте: первый оператор в методе *DoPage* устанавливает в 0 позицию в потоке *MemoryStream*. В противном случае возникает исключение, очень трудное для диагностики.

Создавать метафайл в памяти полезно, если вы хотите передать его через буфер обмена. Передачу метафайлов через буфер обмена мы обсудим в главе *24*.

Ограничивающий прямоугольник

При добавлении в метафайл команд рисования вычисляется ограничивающий прямоугольник, т. е. наименьший прямоугольник, в который можно вписать все объекты, хранящиеся в метафайле. Его ширину и высоту позволяют узнать свойства *Size, Width* и *Height*, наследуемые классом *Metafile* от класса *Image*. А его координаты можно получить из свойства *Bounds* объекта *MetafileHeader*, соответствующего данному метафайлу.

Из представленных в этой главе программ видно, что ограничивающий прямоугольник часто бывает больше, чем требуется. Управлять им позволяют альтернативные конструкторы класса *Metafile*. Вот четыре конструктора, которые устанавливают размер прямоугольника при создании метафайла, хранящегося в файле:

Конструкторы *Metafile* (выборочно)

```
Metafile(string strFileName, IntPtr ipHdc, Rectangle rect)
Metafile(string strFileName, IntPtr ipHdc, RectangleF rectf)
Metafile(string strFileName, IntPtr ipHdc, Rectangle rect,
         MetafileFrameUnit mfu)
Metafile(string strFileName, IntPtr ipHdc, RectangleF rectf,
         MetafileFrameUnit mfu)
```

Перечисление *MetafileFrameUnit* определено в пространстве имен *System.Drawing.Imaging*. Его члены определяют единицы, в которых указан размер ограничивающего прямоугольника. Значение, задаваемое аргументом *MetafileFrameUnit*, в метафайле не сохраняется:

Перечисление *MetafileFrameUnit*

Член	Значение	Комментарии
Pixel	2	Размер в пикселах
Point	3	Размер в 1/72 дюйма
Inch	4	Размер в дюймах
Document	5	Размер в 1/300 дюйма
Millimeter	6	Размер в миллиметрах
GdiCompatible	7	Размер в 1/100 миллиметра

Если аргумент *MetafileFrameUnit* не указывается, по умолчанию применяется *GdiCompatible*). Пожалуй, это не то, что вам нужно!

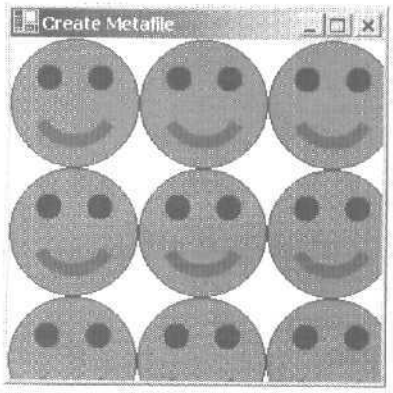
Самой простой единицей измерения при создании простого метафайла являются пикселы. Возьмем, например, оператор из первой версии программы CreateMetafile, создающий метафайл:

```
mf = new Metafile("CreateMetafile.emf", ipHdc);
```

Попробуйте заменить его на:

```
mf = new Metafile("CreateMetafile.emf", ipHdc,
                  new Rectangle(0, 0, 101, 101), MetafileFrameUnit.Pixel);
```

Прямоугольник определен в соответствии с координатами, которые впоследствии будут передаваться методам *Graphics*. Теперь свойства *Width* и *Height* полученного в методе *DoPage* объекта *Metafile* будут иметь значение 101. Таким образом, изображения будут выводиться с интервалом 101 пиксел:



«Почему 101?» — спросите вы. Потому что самый большой объект в данном метафайле был создан таким оператором:

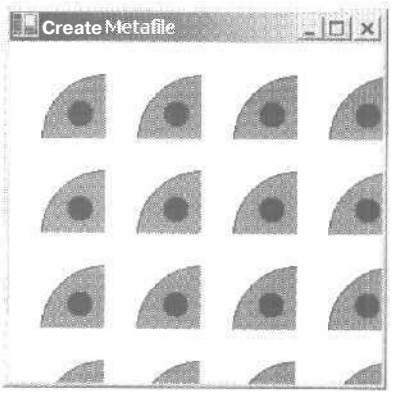
```
grfx.DrawEllipse(Pens.Black, 0, 0, 100, 100);
```

Как вы помните, при ширине пера в 1 пиксел общая длина и ширина такого объекта составит 101 пиксел.

Начальную точку ограничивающего прямоугольника не обязательно помещать в координаты (0, 0). Точное соответствие прямоугольника координатам вывода объектов также не является обязательным. Так, если изменить аргумент, задающий ограничивающий прямоугольник, на:

```
mf = new Metafile("CreateMetafile.emf", ipHdc,
                  new Rectangle(-25, -25, 75, 75), MetafileFrameUnit.Pixel);
```

то рисунки будут отображаться вот так:



Ширина и высота рисунка в метафайле равны 75 пикселам, но начальная точка имеет координаты (-25, -25). Поскольку Б графических объектах метафайла не

использовались отрицательные координаты, верхняя и левая части рисунка пусты. При отображении рисунок отсекается прямоугольником *Bounds*.

Ограничивающий прямоугольник можно задать и в комбинации с аргументом *Stream*:

Конструкторы *Metafile* (выборочно)

```
Metafile(Stream stream, IntPtr ipHdc, Rectangle rect)
Metafile(Stream stream, IntPtr ipHdc, RectangleF rectf)
Metafile(Stream stream, IntPtr ipHdc, Rectangle rect,
         MetafileFrameUnit mfu)
Metafile(Stream stream, IntPtr ipHdc, RectangleF rectf,
         MetafileFrameUnit mfu)
```

Например, можно заменить конструктор из первой версии CreateMetafile на:

```
mf = new Metafile(new MemoryStream(), ipHdc,
                  new Rectangle(0, 0, 101, 101), MetafileFrameUnit.Pixel);
```

Можно создать и новый метафайл, не указывая имени файла или объекта *Stream*:

Конструкторы *Metafile* (выборочно)

```
Metafile(IntPtr ipHdc, Rectangle rect)
Metafile(IntPtr ipHdc, RectangleF rectf)
Metafile(IntPtr ipHdc, Rectangle rect, MetafileFrameUnit mfu)
Metafile(IntPtr ipHdc, RectangleF rectf, MetafileFrameUnit mfu)
```

В этом случае метафайл создается в оперативной памяти, но у вас нет доступа к буферу памяти, как в случае с объектом *MemoryStream*. Вот еще один допустимый оператор создания метафайла для программы CreateMetafile:

```
mf = new Metafile(ipHdc,
                  new Rectangle(0, 0, 101, 101), MetafileFrameUnit.Pixel);
```

Глядя на последний список конструкторов, легко предположить, что можно создать метафайл, указав лишь описатель контекста устройства, но такого конструктора нет.

Разумеется, вы вправе задействовать все значения перечисления *MetafileFrameUnit*, а не только *Pixel*, но это имеет смысл, только если вы создаете рисунок в метафайле, используя другие единицы измерения. Давайте разберемся с влиянием преобразований страниц на метафайл.

Метафайлы и преобразования страниц

Метафайл характеризуется высотой и шириной, узнать которые можно из свойств *Size*, *Width* и *Height* класса *Image* и свойства *Bounds* класса *MetafileHeader*. О ширине и высоте метафайла удобно думать как о его размерах в пикселах, подобно битовой карте. Однако на самом деле эти параметры больше связаны с координатами и размерами, использовавшимися в графических функциях, составляющих метафайл.

Метафайл также имеет разрешение по вертикали и горизонтали, выраженное в точках на дюйм, узнать которое позволяют свойства *HorizontalResolution* и *VerticalResolution* класса *Image* и свойства *DpiX* и *DpiY* класса *MetafileHeader*.

Кроме того, метафайл характеризуется метрическими размерами, которые можно вычислить из соответствующего разрешения и размера в пикселах. Размер в сотых долях миллиметра можно получить из свойства *PhysicalDimension* класса *Image*.

Разрешение метафайла устанавливается при его создании. Конструктору объекта *Metafile* нужно передать описатель контекста устройства, и разрешение этого устройства определит разрешение метафайла. Все координаты и размеры, применяемые в графических функциях метафайла, должны соответствовать этому разрешению. При добавлении в метафайл графических функций на координаты и размеры влияет любое действующее в данный момент преобразование страницы.

Следующая программа создает метафайл с 4 перекрывающимися прямоугольниками. Длина стороны каждого равна 1 дюйму и чертится пером шириной в 1 пт, по при создании каждого прямоугольника используются различные преобразования страницы.

MetafilePageUnits.cs

```
//-----------------------------------------------
// MetafilePageUnits.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Drawing.Printing;     // Пока не используется!
using System.Windows.Forms;

class MetafilePageUnits: PrintableForm
{
     Metafile mf;

     public new static void Main()
     {
          Application.Run(new MetafilePageUnits());
     }
     public MetafilePageUnits()
     {
          Text = "Metafile Psge Units";

               // Создание нового метафайла.

          Graphics grfx  = CreateGraphics();
          IntPtr ipHdc = grfx.GetHdc();

          mf - new Metafile("MetafilePageUnits.emf", ipHdc);

          grfx.ReleaseHdc(ipHdc);
```

см. след. стр.

```
        grfx.Dispose();

            // Получение объекта Graphics для создания рисунка.

        grfx = Graphics.FromImage(mf);
        grfx.Clear(Color.White);

            // Рисование с указанием размеров в пикселах (ширина пера 1 пт),

        grfx.PageUnit = GraphicsUnit.Pixel;
        Pen pen = new Pen(Color.Black, grfx.DpiX / 72);
        grfx.DrawRectangle(pen, 0, 0, grfx.DpiX, grfx.DpiY);

            // Рисование с указанием размеров в 1/100 доима (ширина пера 1 пт).

        grfx.PageUnit = GraphicsUnit.Inch;
        grfx.PageScale = 0.01f;
        pen * new Pen(Color.Black, 100f / 72>;
        grfx.DrawRectangle(pen, 25, 25, 100, 100);

            // Рисование с указанием размеров в миллиметрах (ширина пера 1 пт).

        grfx.PageUnit = GraphicsUnit.Millimeter;
        grfx.PageScale = 1;
        pen = flew Pen(Color.Black, 25.4f / 72);
        grfx.DrawRectangle(pen, 12.7f, 12.7f, 25.4f, 25.4f);

            // Рисование с указанием размеров в 1/72 дюйма (ширина пера 1 пт).

        grfx.PageUnit - GraphicsUnit.Point;
        pen = new Pen(Color.Black, 1);
        grfx.DrawRectangle(pen, 54, 54, 72, 72);

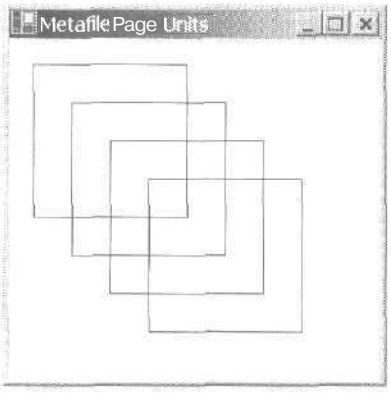
        grfx.Dispose();
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.DrawImage(mf, 0, 0);
    }
K
```

Приятно видеть, что, несмотря на то, что при рисовании четырех прямоугольников их размеры указывались в пикселах, долях дюйма и миллиметрах, они хранятся и отображаются единообразно:

Первый прямоугольник (верхний левый) чертится с указанием размера в пикселах:

```
grfx.PageUnit = GraphicsUnit.Pixel;
Pen pen = new Pen(Color.Black, grfx.DpiX / 72);
grfx.DrawRectangle(pen, 0, 0, grfx.DpiX, grfx.DpiY);
```

По умолчанию значение свойства *PageUnit* для метафайла, созданного на основе разрешения экрана, устанавливается как *GraphicsUnit.Pixel*, так что первый оператор необязателен. Но он нужен, если вы захотите внести изменения, о которых я расскажу ниже,



Все метафайлы, которые мы создавали до сих пор, основывались на разрешении экрана. Можно создать метафайл, соответствующий разрешению принтера. Для этого нужно только заменить оператор:

```
Graphics grfx  = CreateGraphics();
```

в начале кода конструктора следующими операторами, которые получают объект *PrinterSettings*, соответствующий принтеру, используемому по умолчанию, и объект *Graphics* для создания метафайла:

```
PrinterSettings prnset = new PrinterSettings();
Graphics grfx = prnset.CreateMeasurementGraphics();
```

При выводе на экран и на принтер такой метафайл будет выглядеть так же, как исходный, созданный для экрана. Однако при более детальном изучении становится понятно, что разрешение нового метафайла теперь соответствует разрешению вашего принтера, используемого по умолчанию. Кроме того, все координаты и размеры в метафайле соответствуют этому более высокому разрешению,

Типы метафайлов

Метафайл состоит из заголовка и записей. Позже мы узнаем, как просматривать записи метафайла, пока же следует знать, что каждая запись идентифицируется членом перечисления *EmfPlusRecordType*, определенного в пространстве имен *System.Drawing.Imaging*. Это самое большое в .NET Framework перечисление содержит 253 члена. Ниже приведены три взаимосвязанных члена этого перечисления,

Перечисление *EmfPlusRecordType*

Член	Значение
EmfPolyline	4
DrawLines	16397
WmfPolyline	66341

Префикс *Wmf* третьего члена этого небольшого списка расшифровывается как Windows Metafile и соответствует 16-битному формату метафайла. Данная запись указывает на вызов функции *Polyline* графической системы GDI с указанием 16-битных координат. Записи *WmfPolyline* встречаются в метафайлах, созданных до появления 32-битных версий Windows или в метафайлах, создаваемых с учетом обратной совместимости с ранними версиями программ. Такие метафайлы все еще широко применяются в библиотеках иллюстраций.

Член с именем *EmfPolyline* имеет префикс *Emf*, расшифровывающийся как Enhanced Metafile и соответствующий 32-битному формату метафайла. Данная запись также соответствует вызову функции *Polyline,* но указывает на использование 32-битных координат. Такие записи можно встретить в метафайлах расширенного формата, созданных в 32-битных версиях Windows.

Запись *DrawLines* соответствует методу *DrawLines* класса *Graphics*. Такие записи встречаются только в метафайлах формата EMF+, созданных приложениями WindowsForms.

В целях обеспечения обратной совместимости с 32-битными программами Windows класс *Metafile* по умолчанию создает метафайлы, включающие в себя два набора записей: записи EMF (такие как *EmfPolyline*) и EMF+ (например, *DrawLines)*. При этом записи EMF повторяют функции записей EMF+,

Однако если создаваемый метафайл будет применяться лишь в каких-то определенных целях, его размер можно сократить. Например, если вы планируете использовать метафайл только с программами Win32, ему не требуются записи формата EMF+. А если он создается только для программ Windows Forms, то отпадает необходимость в записях EMF.

Вот конструкторы класса *Metafile*, содержащие аргумент *EmfType* и необязательную строку описания:

Конструкторы *Metafile* (выборочно)

```
Metafile(string strFileName, IntPtr ipHdc, EmfType et)
Metafile(string strFileName, IntPtr ipHdc, EmfType et, string strDescription)
Metafile(Stream stream, IntPtr ipHdc, EmfType et)
Metafile(Stream stream, IntPtr ipHdc, EmfType et, string strDescription)
Metafile(IntPtr ipHdc, EmfType et)
Metafile(IntPtr ipHdc, EmfType et, string strDescription)
```

Строка описания обычно содержит сведения о рисунке (например, там можно указать владельца авторских прав) и встраивается в заголовок метафайла. Перечисление *EmfType* состоит из следующих членов:

Перечисление *EmfType*

Член	Значение
EmfOnly	3
EmfPlusOnly	1
EmfPlusDual	5

Для метафайлов, содержащих только EMF-записи, в аргументе *EmfType* задается значение *EmfOnly*. Его следует указывать, только если создаваемый метафайл не будут использовать программы Windows Forms.

Значение *EmfPlusOnl*задастся для метафайлов, содержащих только записи EMF+. Такие метафайлы имеют сравнительно небольшой размер, но могут использоваться только программами Windows Forms,

По умолчанию аргумент *EmfType*принимает значение *EmfPlusDual*. Такой метафайл содержит записи обоих форматов, и, следовательно, его размер примерно равен сумме размеров метафайлов, соответствующих значениям *EmfOnly*и *EmfPlusOnly*.

Чтобы в программе Windows Forms создать 16-битный метафайл формата WMF, требуется при помощи кода Win32 создать описатель WMF и передать его соответствующему конструктору класса *Metafile*. Конструктору *Metafile*также можно передавать описатель EMF, созданный при помощи кода Win32.

К этому моменту я перечислил 22 конструктора из 39 и описал еще 3, создающие объект *Metafile* на основе описателей Win32. Оставшиеся 14 конструкторов позволяют одновременно определять ограничивающий прямоугольник и тип метафайла.

Десяти из них вначале передается имя файла или объект *Stream*, а затем задается аргумент *IntPtr* для описателя контекста устройства. Далее указываетсяобъект типа *Rectangle* или *RectangleF* и один из членов перечисления *MetafileFrame'nit*. И, наконец, можно задать член перечисления *EmfType*и строку описания (или и то и другое вместе). Для конструкторов, которым передается объект *Stream,* указание члена *EmfType* обязательно.

Четыре других конструктора начинаются с указания аргумента *IntPtr* для описателя контекста устройства. Далее следует объект типа *Rectangle* или *RectangleF*, *а* за ним — член перечисления *MetafileFrameUnit*В заключение задается член перечисления *EmfType*.Также можно указать строку описания.

Вот три наиболее обобщенных, полных и длинных конструктора класса *Metafile*:

Конструкторы *Metafile* **(выборочно)**

```
Metafile(string strFlleName, IntPtr ipHdc, RectangleF rectf,
        MetafileFrameUnit mfu, EmfType et, string strDescriptton)
Metafile(Stream stream, IntPtr ipHdc, RectangleF rectf,
        MetafileFrameUnit mfu, EmfType et, string strDescription)
Metafile(IntPtr ipHdc, RectangleF rectf,
        MetafileFrameUnit mfu, EmfType et, string strDescription)
```

Обработка записей метафайла

Поскольку метафайлы часто хранятся на диске, любознательный программист может испытать соблазн отрыть их как обычные файлы и разобраться с их внутренней структурой. Интересно, что класс *Graphics* содержит метод, позволяющий работать с отдельными записями метафайла. Обычно в программе создается отдельный метод, вызываемый при обработке каждой записи,

Существует 36 версий метода *Enumerate Metafile*,самая простая из которых такова;

Методы ***EnumerateMetafile*** **класса** ***Graphics*** **(выборочно)**

```
void EnumerateMetafile(Metafile mf, Point pt,
                       Graphics.EnumerateMetafileProc emp)
```

Принадлежность метода *EnumerateMetafile* к классу *Graphics* может казаться несколько странной, но вскоре вы увидите, что он позволяет не только получать доступ к отдельным записям метафайла, но и последовательно их визуализировать. При считывании каждой следующей записи метафайла ее можно визуализировать, пропустить или даже (при должной смелости) модифицировать перед отображением. Такое разнообразие возможных действий — одна из причин массы разновидностей метода *EnumerateMetafile*. Эти методы идентичны перегруженным операторам *DrawImage*.

Вместо того чтобы перечислять все 36 вариантов метода *EnumerateMetafile*, я привел список необходимых и дополнительных аргументов этого метода.

- В качестве первого аргумента всегда должен указываться объект типа *Metafile*.
- Вторым аргументом всегда определяются координаты вывода. Здесь можно задать значение типа *Point, PointF*, массив из трех элементов типа *Point* или *PointF*, значение типа *Rectangle* или *RectangleF*. Подобно тому, как это было в методе *DrawImage*, при указании массива из трех элементов типа *Point* или *PointF,* они интерпретируются как координаты верхнего левого, верхнего правого и нижнего левого угла изображения.
- Далее следует необязательный аргумент, в котором можно определить исходный прямоугольник в метафайле. Если в предыдущем аргументе применялось значение типа *Point,* массив *Point* или значение типа *Rectangle*, исходный прямоугольник должен быть *Rectangle*. Если координаты вывода указывались значением типа *PointF,* массивом *PointF* или значением *RectangleF,* исходный прямоугольник должен быть *RectangleF*. Далее надо задать значение *GraphicsUnit*, определяющее единицы измерения для исходного прямоугольника.
- Следующий аргумент обязателен. Здесь указывается метод программы, соответствующий делегату *GraphicsEnumerateMetafileProc*. Этот метод вызывается при обработке каждой записи метафайла.
- Далее идет необязательный аргумент *IntPtr*, определенный как указатель на данные, которые должны передаваться в метод, заданный в предыдущем аргументе. Однако аргумента для этих данных в делегате *GraphicsEnumerateMetafileProc* нет,
- Если в вызове метода есть аргумент *IntPtr*, за ним можно указать еще один необязательный аргумент — объект *ImageAttributes*, задающий некоторые особенности отображения рисунка.

Вот как может выглядеть простейший вызов метода *EnumerateMetafile* в реальной программе:

```
grfx.EnumerateMetafile(mf, new Point(0, 0),
                       new Graphics.EnumerateMetafileProc(EnumMetafileProc));
```

Последний аргумент устанавливает ссылку на метод *EnumMetafileProc*, соответствующий делегату *GraphicsEnumerateMetafileProc*. Этот метод может выглядеть как-то так:

```
bool EnumMetafileProc(EmfPlusRecordType eprt, int iFlags,
                      int iDataSize, IntPtr ipData,
                      PlayRecordCallback prc)
{
    ⋮
    return bContinue;
}
```

При необходимости продолжить обработку записей метод возвращает *true,* в противном случае — *false.*

Вскоре я подробнее расскажу об аргументах, передающихся методу обработки записей, а пока лишь коротко опишу их. Первым передается член перечисления *EmfPlusRecordType*указывающий тип записи. Аргумент *iFlags* не документирован.

Аргумент *iDataSize* определяет количество байт данных, на которые ссылается указатель *ipData.* Для каждого из типов записей эти данные различны. Так, для записи, соответствующей ломаной линии, они будут содержать число точек и их координаты.

Последний аргумент имеет тип *PlayRecordCallback* — это делегат, определенный в пространстве имен *System.Drawing.Imaging.*Как вы помните, *EnumerateMetafileProc*тоже является делегатом, а для делегата крайне необычно передавать в числе аргументов еще один делегат. Вот как определяется делегат *PlayRecordCallback:*

```
public delegate void PlayRecordCallback(EmfPlusRecordType eprt, int iFlags,
                                        int iDataSize, IntPtr ipData);
```

Вместо того чтобы определять метод в программе в соответствии с делегатом *PlayRecordCallback* (для чего обычно и применяются делегаты), последний аргумент метода обработки записей метафайла указывает на метод, который должен быть вызван для отображения данной записи метафайла. Делегат определяет аргументы этого метода. Так что метод обработки записей метафайла мог бы выглядеть так:

```
bool EnumMetafileProc(EmfPlusRecordType eprt, int iFlags,
                      int iDataSize, IntPtr ipData,
                      PlayRecordCallback prc)
{
    prc(eprt, iFlags, iDataSize, ipData);
    return true;
}
```

Обратите внимание на вызов делегата *PlayRecordCallback.* При вызове метода *EnumerateMetafile* с обработчиком *EnumMetafileProc* метод *EnumerateMetafile*должен работать в точности, как *DrawImage,* только чуть медленнее, поскольку *EnumMetafileProc*обращается к каждой записи метафайла,

Увы! Делегат *PlayRecordCallback* не работает, и вам, вероятно, придется просто отказаться от последнего аргумента метода обработки записей. Вместо него можно использовать метод класса *Metafile:*

Метод *PlayRecord* класса *Metafile*

```
void PlayRecord(EmfPlusRecordType eprt, int iFlags, int iDataSize,
                byte[] abyData)
```

Этот метод во многом похож на делегат *PlayRecordCallback* за исключением того, что в последнем аргументе вместо указателя *IntPtr* применяется массив байтов. Для преобразования аргумента *IntPtr,* передающегося методу обработки записей, в такой массив, можно использовать статический метод *Copy* класса *Marshal,* определенного в пространстве имен *System.Runtime.InteropServices.*

Поскольку из метода обработки записей вызывается метод *PlayRecord.* принадлежащий классу *Metafile,* объект *Metafile* должен быть сохранен в программы в виде поля. Вот пример метода обработки записей, который просто отображает метафайл на экране.

```
bool EnumMetafileProc(EmfPlusRecordType eprt, int iFlags,
                      int iDataSize, IntPtr ipData,
                      PlayRecordCallback prc)
{
    byte[] abyData = new Byte[iDataSize];
    Marshal.Copy(ipData, abyData, 0, iDataSize);
    mf.PlayRecord(eprt, iFlags, iDataSize, abyData);
    return true;
}
```

Следующая программа создает текстовое поле, доступное только для записи, панель и разделитель между ними. В ней также используется диалоговое окно *OpenFileDialog* для открытия метафайла. Рисунок из метафайла обычным способом выводится на панель. При первой загрузке метафайла программа создает объект *StringWriter* и вызывает метод *EnumerateMetafile* для обработки записей метафайла методом *EnumMetafileProc.* Последний помещает форматированную информацию в объект *StringWriter,* и после возвращения из метода *EnumerateMetafile* получившаяся строка выводится в текстовом поле.

EnumMetafile.cs

```
//-----------------------------------------
// EnuraMetafile.cs © 2001 by Charles Petzold
//                  -      ------      --_---
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Runtime.InteropServices;
using System.Windows.Forms;

class EnumMetafile: Form
{
    Metafile    mf;
    Panel       panel;
```

см. след, стр.

```
TextBox       txtbox;
string        strCaption;
StringWriter strwrite;

public static void Main()
{
     Application.Run(new EnumMetafile());
}
public  EnumMetafile()
{
     Text = strCaption = "Enumerate Metafile";

          // Создание текстового поля для отображения записей.

     txtbox = new TextBox();
     txtbox.Parent = this;
     txtbox.Dock - DockStyle.Fill;
     txtbox.Multiline = true;
     txtbox.WordWrap = false;
     txtbox.ReadOnly - true;
     txtbox.TabStop = false;
     txtbox.ScrollBars = ScrollBars.Vertical;

          // Создание разделителя между текстовым полем и панелью.

     Splitter splitter = new Splitter();
     splitter.Parent = this;
     splitter.Dock = DockStyle.Left;  // Right;

          // Создание панели для отображения рисунка.

     panel = new Panel();
     panel.Parent = this;
     panel.Dock = DockStyle.Left;
     panel.Paint += new PaintEventHandler(PanelOnPaint);

          // Создание мен».

     Menu = new MainMenu();
     Menu.MenuItems.Add("&Open!", new EventHandler(MenuOpenOnClick));
}
void MenuOpenOnClick(object obj, EventArgs ea)
{
     OpenFileDialog dlg = new OpenFileDialog();

     dlg.Filter = "All Metafiles|*.wmf;*.emf|" +
                  "Windows Metafile (*.wmf)|*.wmf|" *
                  "Enhanced Metafile (*.emf)|*.emf";

     if (dlg.ShowDialog() == DialogResult.OK)
```

см. след. стр.

```
        {
            try
            {
                Bf - new Metafile(dlg.FileName);
            }
            catch (Exception exc)
            {
                MessageBox.Show(exc.Message, strCaption);
                return;
            }
            Text - strCaption + " - " + Path.GetFileName(dlg.FileName);
            panel.Invalidate();

                // Отображение записей метафайла в текстовом поле,

            strwrite = new StringWriter();
            Graphics grfx - CreateGraphics();

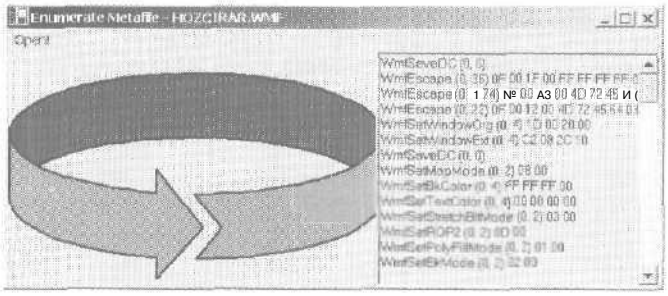
            grfx.EnumerateMetafile(mf, new Point(0, 0),
                new Graphics.EnumerateMetafileProc(EnumMetafileProc));

            grfx.Dispose();
            txtbox.Text * strwrite.ToString();
            txtbox.SelectionLength = 0;
        }
    }
    bool EnumMetafileProc(EmfPlusRecordType eprt, int iFlags,
                          int iDataSize, IntPtr ipData,
                          PlayRecordCallback prc)
    {
        strwrite.Write("{0} ({1}, {2})", eprt, iFlags, IDataSize);

        if (iDataSize > 0)
        {
            byte[] abyData = new Byte[iDataSize];
            Marshal.Copy(ipData, abyData, 0, iDataSize);

            foreach (byte by in abyData)
                strwrite.Write(" {0:X2}", by);
        }
        strwrite.WriteLine();
        return true;
    }
    void PanelOnPaint(object obj, PaintEventArgs pea)
    {
        Panel    panel = (Panel) obj;
        Graphics grfx  = pea.Graphics;
        if (mf != null)
            grfx.DrawImage(mf, 0, 0);
    }
}
```

Хотя эта программа совершенно загадочно не работает ни с одним из метафайлов, созданных программами из этой главы (или из моей книги *«Программирование для Windows"),* вот как она будет выглядеть, если использовать один из метафайлов, поставляющихся в комплекте с Visual Studio .NET:



Справа показаны записи метафайла. Вслед за типом записи в круглых скобках указывается флаг и количество байтов данных. Далее идут сами байты в шестнадцатеричном формате. Преобразование этих данных в более информативный вид в задачи программы не входило.

Как видите, длина записей метафайла непостоянна. Каждый тип записи соответствует определенной функции GDI или методу *Graphics.* Данные каждой записи соответствуют аргументам, передающимся при вызове этих функций или методов. Очевидно, что количество и структура этих данных зависят от типа записи.

На момент написания книги записи метафайла, соответствующие методам класса *Graphics,* оставались недокументированными, но они, конечно, еще будут описаны. Если вы не хотите ждать выхода описания, можете начать с изучения формата предыдущих версий записей для файлов WMF и EMF. Запись каждого типа соответствует структуре, начинающейся с букв *EMR* (enhanced metafile record — расширенный формат записи метафайла), определенной в заголовочных файлах Win32.

Структура EMR является частью каждой записи. В ней указывается тип и размер записи:

```
typedef struct tagEHR
{
     DWORD iType;
     DWORD nSize;
}
EMR;
```

Вот как выглядит структура для ломаной линии. Как и все остальные, она начинается со структуры EMR.

```
typedef struct tagEMRPOLYLINE
{
     EMR emr;
     RECTL rclBounds;
     DWORD cptl;
     POINTL aptl[1];
}
EMRPOLYLINE;
```

Все, что идет за полем EMR, соответствует данным, составляющим запись для Windows Forms,

Если вы не знакомы со структурами Win32, учтите, что поля *RECTL* и *POINTL* хранят значения типа *long* (но речь идет не о 64-битном типе C#, а о 32-битном типе языка C), а поля *RECTS* и *POINTS* — значения типа *short,* длиной 16 бит. Кроме того, прямоугольник определяется не координатами верхнего левого угла и значениями ширины и высоты, а координатами верхнего левого и нижнего правого углов.

ГЛАВА

24



Буфер обмена и буксировка

Буфер обмена Windows (Windows clipboard) позволяет передавать данные из одного приложения в другое. Это относительно простой механизм, не требующий больших издержек ни от программы, помещающей в него данные, ни от программы, которая потом к ним обращается. Большинство приложений использует его, даже когда операции вырезания и вставки связаны с передачей данных внутри приложения,

Программы, работающие с документами или другими данными, предоставляют пользователю доступ к буферу обмена через стандартные пункты меню Cut (Вырезать), Copy (Копировать) и Paste (Вставить). Эти пункты дублируют быстрые клавиши, унаследованные от Apple Macintosh: Ctrl+X, Ctrl+C и Ctrl+V. Когда пользователь выбирает Cut или Copy, программу передает выбранные данные из приложения в буфер обмена. Данные имеют определенный формат или форматы: обычно это текст, растровое изображение, метафайл или двоичные данные, Команда Cut, кроме того, удаляет выделенные данные из документа. Когда пользователь выбирает в меню пункт Paste, программа определяет, содержатся ли в буфере обмена данные, с которыми она может работать. Если да, то они передаются из буфера в программу.

Программа не должна передавать данные в буфер обмена без явного указания со стороны пользователя: выполняя операцию Cut или Copy (Ctrl+X или Ctrl+C), он вправе рассчитывать, что данные останутся в буфере обмена до следующей операции вырезания или копирования

Как вы, наверное, помните по главе 18, элементы управления *TextBox* и *RichTextBox* имеют собственные интерфейсы с буфером обмена. Программе, использующей эти элементы, нужно только лишь вызывать соответствующие методы, реализованные в *TextBoxBase*. Однако в общем случае этой удобной возможности у вас нет. Вы должны самостоятельно обращаться к буферу обмена.

Возможность буксировки в Windows тесно связана с буфером обмена, поэтому я также рассмотрю ее в этой главе.

Элементы и форматы

За один раз в буфер обмена можно записать только один элемент. Вновь копируемый элемент заменяет тот, что находился в буфере раньше.

Однако приложение может копировать конкретный элемент в буфер обмена в разных форматах. Например, рассмотрим программу работы с электронными таблицами, в которой пользователь выделяет массив строк и столбцов, а затем выбирает команду Copy. Программа работы с электронными таблицами, вероятно, будет использовать различные форматы для записи этих строк и столбцов в буфер обмена. Пожалуй, для приложения наиболее важен двоичный формат, известный только самому приложению: он позволяет программе восстановить в точности такие же данные (с любым форматированием), когда пользователь выполнит команду Paste.

Программа работы с таблицами, кроме того, может скопировать выделенные строки и столбцы в буфер обмена в текстовом формате, разделенном запятыми, который смогут использовать другие программы работы с таблицами или базами данных. Программа также может отформатировать данные как текст, в котором между столбцами находятся символы табуляции; такой формат хорошо подходит для вставки данных в программу обработки текстов. Наверное, программа работы с электронными таблицами может даже создать растровое изображение или метафайл, содержащий изображение строк и столбцов,

Приложению, вставляющему данные из буфера обмена, не требуется поддерживать все эти форматы — оно может просто выбрать наиболее эффективный формат.

Логично предположить, что существование элементов буфера обмена в различных форматах на порядок усложняет программирование работы с буфером обмена.

Очень маленький (но мощный) класс *Clipboard*

Класс *Clipboard* входит в пространство имен *System.Windows.Forms*. Этот класс изолированный, и у него нет открытых конструкторов. Нельзя создать экземпляр данного класса и нельзя унаследовать от него другой класс. У класса нет свойств — есть лишь два статических метода, один из которых имеет две версии. Эти методы служат для помещения данных в буфер обмена или получения данных из буфера обмена:

Статические методы *Clipboard*

```
void SetDataObject(object obj)
void SetDataObject(object ob], bool bRemainAfterExit)
IDataObject GetDataObject()
```

Пока вы работаете только с одним форматом данных, из этих двух более простой операцией является помещение данных в буфер. Так, если у вас есть объект *str* типа *string* и вы хотите скопировать его в буфер обмена, просто напишите:

```
Clipboard.SetDataObject(str, true);
```

Данные, которые перед этим были в буфере обмена, просто будут удалены и заменены на этот элемент. Аналогично можно поместить в буфер растровое изображение:

```
Clipboard.SetDataObject(bitmap, true);
```

Кроме того, в буфер обмена можно поместить метафайл:

```
Clipboard.SetDataObject(metafile, true);
```

Во всех этих трех случаях для того, чтобы буфер обмена мог работать с объектом, делается копия объекта. Если изменить объект после вызова *SetDataObject,* не произойдет ничего страшного. Такая последовательность операторов не вызывает никаких проблем:

```
Clipboard.SetDataObject(str, true);
str = null;
```

Единственный способ повлиять на объект, оказавшийся в буфере обмена, — еще раз вызвать *SetDataObject,* и тогда новый элемент заменит старый.

В этих примерах я присвоил *true* второму параметру *SetDataObject,* так как именно это значение вам скорее всего придется использовать при помещении строк, растровых изображений или метафайлов в буфер обмена. Если присвоить второму параметру *false*:

```
Clipboard.SetDataObject(str, false);
```

или вообще убрать этот параметр:

```
Clipboard.SetDataObject(str);
```

то элемент, помещенный в буфер обмена, исчезнет по завершении программы. Пожалуй, для пользователя лучше, когда элемент остается в буфере обмена независимо от того, выполняется программа или нет,

Однако элементы, помещаемые в буфер, не ограничиваются строками, растровыми изображениями и метафайлами. В буфер обмена можно помещать *любой* объект. Вот код, создающий новый объект *Button*, присваивающий значение свойству *Text* и затем копирующий объект в буфер:

```
Button btn = new Button();
btn.Text = "OK";
Clipboa rd.SetDataObject(btn);
```

Когда в *SetDataObject* передается объект, не являющийся строкой, метафайлом или растровым изображением, нужно использовать сокращенную *форму SetDataObject* или указывать *false* в качестве второго параметра. Дело в том. что буфер обмена не может передавать произвольные объекты (вроде *Button*) между приложениями. Только приложение, поместившее объект *Button* в буфер, сможет прочитать этот объект. Так что нет смысла оставлять объект в буфере обмена по завершении программы. Как я уже говорил, проще всего помещать данные в буфер обмена, если ограничиваться одним форматом. Ниже я покажу, как помещать в буфер данные в нескольких форматах.

Взять данные из буфера обмена не так просто, как положить. Метод *GetDataObject* определен как возвращающий экземпляр класса, реализующего интерфейс *IDataObject,* т. е. *GetDataObject* вызывается так:

```
IDataObject data = Clipboard.GetDataObject()
```

После этого объект *data* содержит все, что вам нужно знать о содержимом буфера обмена. Теперь мы должны сосредоточиться на интерфейсе *IDataObject.*

Получение объектов из буфера обмена

Для начала я собираюсь показать довольно простой и прямолинейный способ получения данных из буфера обмена — действия, которое вы скорее всего будете выполнять при выборе пользователем пункта меню Paste.

Статический метод *GetDataObject* класса *Clipboard* не описан в документации как возвращающий объект определенного класса — только как возвращающий экземпляр класса, реализующего интерфейс *IDataObject.* Этих сведений достаточно, чтобы использовать методы *IDataObject* объекта, возвращаемого *GetDataObject.* В *IDataObject* определяются 4 метода (12, если учитывать перегрузки). Вот два из них в самой простой форме:

Методы *IDataObject* (выборочно)

```
bool GetDataPresent(Type type)
object GetData(Type type)
```

Метод *GetDataPresent,* пожалуй, правильнее было бы назвать *IsDataPresent* для большей совместимости с остальными методами и свойствами Windows Forms. Если у вас имеется объект *data,* возвращаемый *Clipboard.GetDataObject,* то выражение:

```
data.GetDataPresent(typeof(String))
```

возвращает *true,* если к буфере обмена содержится объект *String.* Обратите внимание, как оператор *typeof* применяется для получения объекта *Type,* идентифицирующего класс *String.* Аналогично:

```
data.GetDataPresent(typeof(Bitmap))
```

возвращает *true,* если в буфере обмена находится объект *Bitmap,* а:

```
data.GetDataPresent(typeof(Metafile))
```

возвращает *true,* если доступен объект *Metafile.* Если у вас есть привычка помещать в буфер обмена нестандартные объекты, то можно использовать вызовы:

```
data.GetDataPresent(typeof(Button))
```

Их можно осуществлять, не сохраняя значение, возвращаемое *Clipboard.GetDataObject.* Например:

```
Clipboard.GetDataObject().GetDataPresent(typeof(Bitmap))
```

возвращает *true,* если буфер обмена содержит объект *Bitmap.*

Часто метод *GetDataPresent* применяется в событии *Popup* меню Edit (Правка). Пункт Paste при этом следует активизировать, только когда *GetDataPresent* возвращает *true* для интересующего вас типа данных.

Кстати, из того, что:

```
data.GetDataPresent(typeof(String»
```

возвращает *true,* не следует, что для других типов *GetDataPresent* возвратит *false.* Как я уже говорил, буфер обмена может содержать несколько форматов для одного элемента. В буфере могут находиться объект *String* с текстом, *объект Metafile,* содержащий вызов *DrawString,* показывающий ту же самую строку, и прорисовка того же текста в объекте *Bitmap.* Приложению, получающему данные из буфера обмена, следует определять наиболее полезный формат.

Получить объект из буфера обмена позволяет метод *GetData,*

```
string str = (string) data.GetData(typeof(string));
```

Так как *GetData* возвращает *object,* возвращаемое значение нужно привести к требуемому типу данных. Аналогично:

```
Bitmap bitmap = (Bitmap) data.GetData(typeof(Bitmap));
```

Получение объекта из буфера обмена не влияет на содержимое буфера обмена. Возвращаемое *GetData* значение — это копия объекта, хранящегося в буфере обмена.

Посмотрим, как это работает. В следующей программе делается немного больше, чем копирование объектов *String* в буфер обмена и их получение из него.

ClipText.cs

```
//-----------------------------------------------
// ClipText.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ClipText: For»
{
    string   strText = "Sample text for the clipboard";
    MenuItem miCut, miCopy, miPaste;

    public static void Main()
    {
        Application.flun(newClipTextO);
    }
    public ClipTextO
    {
        Text = "Clip Text";
        ResizeRedraw * true;
```

см. след. стр.

```
        Menu = new MainMenu();

            // Меню Edit

        MenuItem Hi = new MenuItem("&Edit");
        mi.Popup += new EventHandler(MenuEditOnPopup);
        Menu.MenuItems.Add(mi);

            // Пункт Cut меню Edit
        miCut = new MenuItem("Cu&t");
        miCut.Click += new EventHandler(MenuEditCutOnClick);
        miCut.Shortcut = Shortcut.CtrlX;
        Menu.MenuItems[0].MenuItems.Add(miCut);

            // Пункт Copy меню Edit

        roiCopy - new MenuItem("&Copy");
        miCopy.Click += new EventHandler(MenuEditCopyOnClick);
        miCopy.Shortcut в Shortcut.CtrlC;
        Menu.MenuItems[0].MenuItems.Add(miCopy);

            // Пункт Paste мен» Edit

        miPaste = new MenuItem("&Paste");
        miPaste.Click += new EventHandler(MenuEditPasteOnClick);
        miPaste.Shortcut = Shortcut.CtrlV;
        Menu.MenuItems[0].MenuItems.Add(miPaste);
    }
    void MenuEditOnPopup(object obj, EventArgs ea)
    {
        miCut.Enabled =
        miCopy.Enabled = strText.Length > 0;
        miPaste.Enabled *
                Clipboard.GetDataObject().GetDataPresent(typeof(string));
    }
    void MenuEditCutOnClick(object obj, EventArgs ea)
    {
        MenuEditCopyOnClick(obj, ea);
        strText = "";
        Invalidate();
    }
    void MenuEditCopyOnClick(object obj, EventArgs ea)
    {
        Clipboard.SetDataObject(strText, true);
    }
    void MenuEditPasteOnClick(object obj, EventArgs sa)
    {
        IDataObject data = Clipboard.GetDataObject();

        if (data.GetDataPresent(typeof(string)))
```

см. след. стр.

```
                strText = (string) data.GetData(typeof(string));

            Invalidate();
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics grfx = pea.Graphics;
            StringFormat strfmt = new StringFormat();
            strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;

            grfx.DrawString(strText, Font, new SolidBrush(ForeColor),
                            ClientRectangle, strfmt);
        }
    }
```

Программа ClipText содержит переменную *strText* типа *string*, показываемую в центре клиентской области. В конструкторе создается меню Edit с пунктами Cut, Copy и Paste. В обработчике события *Popup* пункты Cut и Copy активизируются, только если длина строки ненулевая. Пункт Paste активизируется, только если буфер обмена содержит объект *string*.

В обработчике события *Click* команды Copy для копирования *strText* в буфер обмена служит метод *SetDataObject* класса *Clipboard*. В обработчике события Cut вызывается обработчик события Copy, и, кроме того, строка удаляется из программы путем присваивания *strText* пустой строки.

В обработчике события *Click* команды Paste сначала проверяется, содержится ли в буфере объект типа *string*. (Вы можете посчитать эту проверку избыточной, так как если в буфере обмена не хранится текст, то и пункт меню Paste не будет активен. Но нужно иметь в виду, что Windows — многозадачная среда, и возможно, что содержимое буфера изменится между показом подменю и щелчком пункта меню. Вызов *GetData* для типа объекта, которого более нет в буфере обмена, не приведет к генерации исключения, но *GetData* возвратит значение *null*, а программа не совсем готова к этой ситуации.) Если буфер обмена содержит текст, в обработчике события Paste вызывается метод *GetData* для получения объекта *string*, затем эта строка присваивается *strText*.

Можно посмотреть, как ClipText взаимодействует с Microsoft Notepad (Блокнот), текстовыми процессорами и Web-браузерами. Вы увидите, что при копировании текста из текстового процессора или Web-браузера и вставке в ClipText, текст теряет форматирование. Это не является неожиданностью: как вы знаете, объекты *string* обычно не содержат форматирование, и было бы удивительно, если б ClipText показала блок форматированного текста (rich text format, RTF) или тэги HTML, Но скоро я расскажу как при работе с буфером обмена выйти за рамки простого текста.

В главе 23 я показывал программу MetafileViewer и ее наследницу MetafileConvert, которая содержит меню Edit, позволяющее передавать метафайлы в буфер обмена и получать их из него.

MetafileClip.cs

```
//-----------------------------------------
// MetafileClip.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Forms;

class MetafileClip: MetafileConvert
{
     MenuItem miCut, miCopy, miPaste, miDel;

     public new static void Main()
     {
          Application.ftN№(new MetafileClip());
     }
     public MetafileClip()
     {
          Text = strProgName » "Metafile Clip";

               // Меню Edit

          Menu.MenuItems[1].Popup += new EventHandler(MenuEditOnPopup);

               // Пункт Cut меню Edit

          miCut = new MenuItem("Cu&t");
          miCut.Click += new EventHandler(MenuEditCutOnClick);
          miCut.Shortcut = Shortcut.CtrlX;
          Menu.MenuItems[1].MenuItems.Add(miCut);

               // Пункт Copy меню Edit

          miCopy - new MenuItem("&Copy");
          miCopy.Click += new EventHandler(MenuEditCopyOnClick);
          miCopy.Shortcut = Shortcut.CtrlC;
          Menu.MenuItems[1].MenuItems.Add(miCopy);

               // Пункт Paste меню Edit

          miPaste - new MenuItem("&Paste");
          miPaste.Click += new EventHandler(MenuEditPasteOnClick);
          raiPaste.Shortcut= Shortcut.CtrlV;
          Menu.MenuItems[1].MenuItems.Add(miPaste);

               // Пункт Delete меню Edit

          ralOel = new MenuItem("De&lete");
```

см. след. стр.

```
        miDel.Click += new EventHandler(MenuEditDelOnClick);
        miDel.Shortcut = Shortcut.Del;
        Menu.MenuItems[1].MenuItems.Add(miDel);
    }
    void MenuEditOnPopup(object obj, EventArgs ea)
    {
        miCut.Enabled =
        miCopy.Enabled *
        miDel.Enabled = mf != null;
        miPaste.Enabled =
            Clipboard.GetDataObject().GetDataPresent(typeof(Metafile));
    }
    void MenuEditCutOnClick(object obj, EventArgs ea)
    {
        MenuEditCopyOnClick(obj, ea);
        MenuEditDelOnClick(obj, ea);
    }
    void MenuEditCopyOnClick(object obj, EventArgs ea)
    {
        Clipboard.SetDataObject(mf, true);
    >
    void MenuEditPasteOnClick(object obj, EventArgs ea)
    {
        IDataObject data = Clipboard.GetDataObject();

        if (data.GetDataPresent(typeof(Metafile)))
            mf * (Metafile) data.GetData(typeof(Metafile));

        strFileName = "clipboard";
        Text = strProgName + "   " + strFileName;
        Invalidate();
    }
    void MenuEditDelOnClick(object obj, EventArgs ea)
    {
        mf = null;
        strFileName * null;
        Text = strProgName;
        InvalidateO;
    }
}
```

Кроме пунктов Cut, Copy и Paste, меню Edit программы содержит пункт Delete. [В некоторых приложениях он называется Clear (Очистить).] В отличие от Cut пункт Delete не влияет на буфер обмена и удаляет данные без предварительного копирования в буфер. Если Cut и Copy уже реализованы, добавить Delete обычно просто. Действительно, можно рассматривать операцию Cut как операцию Copy, после которой выполняется Delete. Именно так и написан обработчик события *Click* пункта меню Cut:

```
void MenuEditCutOnClick(object obj, EventArgs ea)
{
     MenuEditCopyOnClick(obj, ea);
     MenuEditDelOnClick(obj, ea);
}
```

Так как MetafileClip работает и с файлами, и с буфером обмена, в ней несколько по-другому формируется заголовок формы. В предыдущих версиях программы я присваивал свойству *Text* формы хранящееся в поле *strProgName* имя программы (разбитое на слова), соединенное с именем загруженного в данный момент файла, хранящимся в поле *strFileName:*

```
Metafile Viewer - Picture.emf
```

В программе MetafileClip строка *strProgName* имеет значение «Metafile Clip».

При выборе пункта меню Delete переменной *strFileName* присваивается значение *null,* а свойству *Text* — *strProgName.* Это и понятно. Более сложная проблема: что делать при загрузке метафайла из буфера обмена? Я решил присваивать полю *strFileName* значение «clipboard». Можно было бы присваивать значения «untitled» или «metafile».

В главе 16 я демонстрировал программу ImageOpen, загружающую из файлов объекты Image, и ImageIO, сохраняющую эти объекты. В главе 21 я добавил в программе ImagePrint возможность печати. Теперь в программе ImageClip, унаследованной от ImagePrint, я добавлю возможность работы с буфером обмена.

ImageClip.cs

```
//---------------------------------------
// ImageClip.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Forms;

class ImageClip: ImagePrint
{
     MenuItem miCut, miCopy, miPaste, miDel;

     public new static void Main()
     {
          Application.Run(new ImageClip());
     }
     public ImageClip()
     {
          Text = strProgName = "Image Clip";

               // Меню Edit

          MenuItem mi = new MenuItem("&Edit");
          mi.Popup += new EventHandler(MenuEditOnPopup);
```

см. след. стр.

```
        Menu.MenuItems.Add(mi);
        int index = Menu.MenuItems.Count - 1;

            // Пункт Cut меню Edit

        miCut * new MenuItem("Cu&t");
        miCut.Click += new EventHandler(MenuEditCutOnClick);
        miCut.Shortcut = Shortcut.CtrlX;
        Menu.MenuItems[index].MenuItems.Add(miCut);

            // Пункт Copy меню Edit

        miCopy = new MenuItem("&Copy");
        miCopy.Click += new EventHandler(MenuEditCopyOnClick); ='
        miCopy.Shortcut = Shortcut.CtrlC;
        Menu.MenuItems[index].MenuItems.Add(miCopy);

            // Пункт Paste меню Edit

        miPaste = new MenuItem("&Paste");
        miPaste.Click += new EventHandler(MenuEditPasteOnClick);
        miPaste.Shortcut * Shortcut.CtrlV;
        Menu.MenuItems[index].MenuItems.Add(miPaste);

            // Пункт Delete меню Edit

        ftiDel = new MenuItem("De&lete");
        miDel.Click += new EventHandler(MenuEditDelOnClick);
        miDel.Shortcut = Shortcut.Del;
        Menu.MenuItems[index].MenuItems.Add(miDel);
    }
    void MenuEditOnPopup(object obj, EventArgs ea)
    {
        miCut.Enabled =
        miCopy.Enabled =
        miDel.Enabled = image I- null;

        IDataObject data - Clipboard.GetDataObject();

        miPaste.Enabled = data.GetDataPresent(typeof(Bitmap)) ||
                          data.GetDataPresent(typeof(Metafile));
    }
    void MenuEditCutOnClick(object obj, EventArgs ea)
    {
        MenuEditCopyOnClick(obj, ea);
        MenuEditDelOnClick(obj, ea);
    }
    void MenuEditCopyOnClick(object obj, EventArgs aa)
    {
        Clipboard.SetDataObject(image, true);
```

см. след. стр.

```
    }
    void MenuEditPasteOnClick(object obj, EventArgs ea)
    {
        IDataObject data = Clipboard.GetDataObject();

        if (data.GetDataPresent(typeof(Metafile)))
            image = (Image) data.GetData(typeof(Metafile));

        else if (data.GetDataPresent(typeof(Bitmap)))
            Image = (Image) data.GetData(typeof(Bitmap));

        strFileName = "Clipboard";
        Text = strProgName + " - " + strFileName;
        Invalidate();
    }
    void MenuEditDelOnClick(object obj, EventArgs ea)
    {
        image * null;
        strFileName = null;
        Text = strProgName;
        Invalidate();
    }
}
```

Усовершенствования этой программы аналогичны усовершенствованиям MetafileClip, но ImageClip несколько сложнее. Она работает с объектом *Image,* который хранится в поле *image* и может быть объектом или типа *Bitmap,* или типа *Metafile.* При использовании объекта *Image* в методе *SetDataObject* проблем не возникает;

```
Clipboard.SetDataObject(image, true);
```

Вероятно, метод *SetDataObject* вызывает *GetType* для первого параметра, чтобы определить тип объекта. В зависимости от типа объекта *image GetType* вернет или *System.Drawing.Bitmap,* или *System.Drawing.Imaging.Metafile.*

Однако нельзя использовать *typeof(Image)* в методе *GetDataPresent* или *GetData* интерфейса *IDataObject.* Если программа должна вставлять или растровое изображение, или метафайл, следует активизировать пункт меню Paste, когда буфер обмена содержит *или* растровое изображение, *или* метафайл:

```
miPaste.Enabled = data.GetDataPresent(typeof(Bitmap)) ||
                  data.GetDataPresent(typeof(Metafile));
```

Возможно, что буфер обмена содержит и *Bitmap,* и *Metafile,* являющиеся одним и тем же изображением. При выборе пользователем пункта меню Paste программа должна определить, какой объект загружать.

Я решил предоставить приоритет объекту *Metafile:*

```
if (data.GetDataPresent(typeof(Metafile)))
    image = (Image) data.GetData(typeof(Metafile));
```

```
else if (data.GetDataPresent(typeof(Bitmap)))
    image = (Image) data.GetData(typeof(Bitmap));
```

Мой выбор не случаен! Спросите себя: какие программы помещают в буфер обмена и метафайл, и растровое изображение? Вероятно, это программы векторной графики, по сути работающие с метафайлами. Когда такое приложение копирует изображение в буфер обмена, изображение заодно преобразуется в растровое изображение. Благодаря этому изображение становится доступным приложениям, неспособным работать с метафайлами.

Очень маловероятно, что метафайл поступает из программы растровой графики. Программы растровой графики обычно не умеют преобразовывать растровые изображения в метафайлы. Такие преобразования весьма специфичны и возможны только для очень простых изображений,

Таким образом, метафайл — это *настоящее* изображение, а растровое изображение — лишь результат преобразования. В программе, работающей и с метафайлами, и с растровыми изображениями, следует в первую очередь пытаться загрузить из буфера обмена метафайлы и только во вторую — растровые изображения.

Форматы данных буфера обмена

Сначала все кажется простым: вы помещаете в буфер обмена объект типа *String, Bitmap, Metafile* или даже *Button* и извлекаете из него объект типа *String, Bitmap, Metafile* или *Button,*

Но не все Windows-приложения используют Windows Forms! Некоторые из них помещают в буфер обмена объекты, не отображаемые напрямую в типы Windows Forms. У проблемы есть и другая сторона: Windows-приложения, использующие Win32 API или MFC, не готовы напрямую работать с объектами Windows Forms.

Рассмотрим якобы простой тип данных — *текст.* Некоторые Windows-программы хранят текст в формате Unicode, но большинство этого формата не использует. Если Windows-программа передает в буфер обмена 8-битные строки ANSI, то программа, использующая Windows Forms, должна уметь прочитать этот текст в обычный объект *String.* Аналогично, когда программа, использующая Windows Forms, помещает объект *String* в буфер обмена, эту строку должны читать программы, знакомые только с ANSI. Есть и еще одна кодировка текста — OEM, восходящая к символьной среде MS-DOS. OEM расшифровывается как «original equipment manufacturer» (производитель комплексного оборудования), но в США это на самом деле означает 8-битный набор символов IBM, применявшийся на самых первых ПК. (Возможно, вы помните, символы рисования линий, используемые программами, работающими в текстовом режиме.) Кодировки ANSI и OEM отличаются верхними 128 символами.

Чтобы выполнить эти требования, буфер обмена должен выполнять преобразования между различными текстовыми кодировками. Независимо от того, в какой кодировке текст поступает в буфер обмена, буфер должен создавать представления этого текста в нескольких других кодировках, доступные другим приложениям.

Понятно, что эти преобразования между различными кодировками действительно выполняются. Программа ClipText успешно обменивается текстом с обычными программами Windows, использующими кодировку ANSI или Unicode.

Однако связанные с текстом проблемы не ограничиваются только кодировкой. Рассмотрим случай, когда текстовый процессор копирует блок текста в буфер обмена. Что будет с форматированием текста? Если пользователь захочет вставить этот текст в ту же программу обработки текстов (в тот же самый документ или в другой), то, конечно, текст не должен потерять форматирование. С другой стороны, если пользователь хочет вставить текст в Notepad, следует убрать все форматирование из текста. А что должно произойти при вставке текста в *другую* программу обработки текстов? Скорее всего вы захотите сохранить форматирование, значит, надо записывать текст в буфер обмена, используя форматирование, не зависящее от приложения (например, RTF или, возможно, HTML).

Или допустим, что текст копируется в буфер обмена из Web-браузера. Форматирование HTML следует сохранить для приложений, понимающих HTML, но не стоит применять это форматирование для приложений, не работающих с HTML, Большинство пользователей не обрадуется, увидев тэги HTML при копировании текста из Web-браузера в Notepad!

Некоторые виды текста предназначены для чтения программами, а не людьми. Так, многие программы работы с БД и электронными таблицами могут копировать и вставлять информацию в текстовом формате CSV (comma-separated value, значения, разделенные запятыми). Этот формат обеспечивает совместное использование записей БД или таблиц, не зависящее от приложения,

Так что простое копирование текстовой строки в буфер обмена:

```
Clipboard.SetDataObject(str);
```

вероятно, делает не совсем то, что вам нужно, если строка содержит текст в форматах RTF, HTML или CSV. Лучше всего, если б приложение указывало, какой тип текста на самом деле помещается в буфер обмена.

Когда приложение обработки текстов копирует текст в буфер обмена, наверное, лучше всего преобразовывать внутренний формат приложения в RTF и простой, неформатированный текст в самом приложении. Значит, приложение должно уметь за один раз помещать данные в буфер обмена в нескольких форматах. (Это может показаться невозможным. При вызове *SetDataObject* объект, передаваемый в качестве параметра, замещает текущее содержимое буфера обмена. Нельзя вызвать *SetDataObject* несколько раз, чтобы поместить несколько форматов одного текста в буфер обмена. Однако вы скоро узнаете, что параметром *SetDataObject* может быть объект, который в свою очередь ссылается на несколько разных объектов.)

В программах для Windows Forms растровые изображения и метафайлы — это объекты типов *Bitmap* и *Metafile*. Однако в программах, не использующих Windows Forms, встречаются два типа растровых изображений: аппаратно зависимые растровые изображения старого стиля, появившиеся в Windows 1.0, и DIB (device-independent bitmap, аппаратно независимое растровое изображение), появившиеся в Windows 3.0. Кроме того, как я говорил в главе 23, программы, не использующие Windows Forms, работают с первоначальным форматом метафайлов (WMF) и усовершенствованным форматом метафайлов (EMF).

Вот ключ к решению этой проблемы: в буфере обмена хранятся не только фрагменты данных, но и средства идентификации их форматов. У программ для Windows Forms конкретный формат буфера обмена обычно идентифицируется строкой, например «DeviceIndependentBitmap» или «Text». С этим строками, кроме того, связаны идентификационные номера, например, 8 для «DeviceIndependentBitmap» и 1 для «Text». Эти идентификационные номера — то же самое, что идентификаторы, начинающиеся с *CF* («clipboard format»), определенные в файлах заголовков Win32.

Метод *GetFormats*, определенный в интерфейсе *IDataObject*, предоставляет список всех форматов данных, хранящихся в данный момент в буфере обмена:

Методы *GetFormats* интерфейса *IDataObject*

```
string[] GetFormats()
string[] GetFormats(bool bIncludeConversions)
```

Вызов:

```
string[] astr = data.GetFormats();
```

равносилен:

```
string[] astr = data.GetFormats(true);
```

Оба возвращают строковые обозначения всех форматов буфера обмена, которые находятся в нем в данный момент. Некоторые из этих форматов являются преобразованиями данных, хранящихся в буфере. Чтобы ограничиться списком непреобразованных форматов, используйте:

```
string[] astr = data.GetFormats(false);
```

Иногда данные преобразовываются при помещении в буфер обмена. Такие форматы возвращаются независимо от значения параметра *GetFormats*.

Так, если программа для Windows Forms поместит в буфер обмена объект *String*, то *GetFormats* (независимо от параметра) возвратит:

```
System.String
UnicodeText
Text
```

Типы «UnicodeText* и «Text» позволяют программам, использующим Win32 API или MFC, получить текст из буфера обмена в Unicode или 8-битной кодировке ANSI.

Когда Notepad помещает текст в буфер обмена, *GetFormats* с параметром *false* возвращает такие четыре строки:

```
UnicodeText
Locale
Text
OEMText
```

При вызове *GetFormats* без параметров или с параметром *true* возвращаются эти же четыре строки, а также еще одна:

```
System.String
```

Когда программа, использующая Windows Forms, помещает объект *Bitmap* в буфер обмена, *GetFormats* независимо от параметра возвращает:

```
System.Drawing.Bitmap
WindowsForms10PersistentObject
Bitmap
```

Когда программа для Windows Forms помещает в буфер обмена объект *Metafile*, то независимо от параметра *GetFormats* возвращает:

```
System.Drawing.Imaging.Metafile
WindowsForms10PersistentObject
```

Кроме того, можно использовать строки, обозначающие формат, в методах *GetDataPresent* и *GetData* интерфейса *IDataObject*.

Методы *IDataObject* (выборочно)

```
bool GetDataPresentCstring strFormat)
bool GetDataPresent(string strFormat, bool bAllowConversions)
object GetData(string strFormat)
object GetDataCstring strFormat, bool bAllowConversions)
```

Вызов:

```
string str = (string) data.GetData(typeof(string))
```

равносилен:

```
string str = (string) data.GetData("System.String");
```

Это вполне очевидно. Но, кроме того, эти два вызова равносильны следующему:

```
string str = (string) data.GetData("UnicodeText");
```

Метод *GetData* всегда преобразовывает данные из буфера обмена в тип данных .NET Framework, поэтому в этом случае *GetData* возвращает объект *string*.

Рассмотрим вызов:

```
string str = (string) data.GetData("Text");
```

Он тоже равносилен предыдущим? Может, да, а может, и нет. Если программа, работающая с Unicode, помещает в буфер обмена строку, содержащую, скажем, еврейские или арабские символы или кириллицу, то строка, идентифицируемая строкой «Text», будет 8-битной ANSI-версией исходного текста Unicode. Символы Unicode, не имеющие эквивалента среди символов ANSI, заменятся на вспросительные знаки.

Так или иначе, если присвоить второму параметру *GetDataPresent* или *GetData* значение *false,* получить преобразованный тип не удастся. Вызов:

```
(string) data.GetData("Text", false)
```

возвращает *null,* если в буфере обмена находится текст Unicode,

Если вы избегаете жесткого кодирования строк в программе, можно задействовать класс *DataFormats*. Он содержит 21 статическое неизменяемое свойство, воз-

вращающее строки для форматов буфера обмена, напрямую поддерживаемых .NET Framework. Вот форматы буфера обмена, происходящие из Win32 API. В столбце справа показан идентификационный номер Win32, соответствующий формату.

Статические поля *DataFormats* (выборочно)

Тип	Поле	Значение	Идентификационный номер
siring	*Text*	«Text»	1
string	*Bitmap*	«Bitmap»	2
string	*MetafilePict*	«MetaFilePict»	3
string	*SymbolicLink*	«SymbolicLink»	4
string	*Dif*	«DataInterchangeFormat»	5
string	*Tiff*	«TaggedImageFileFormat»	6
string	*OemText*	«OEMText»	7
string	*Dib*	«DeviceIndependentBitmap»	8
string	*Palette*	«Palette»	9
string	*PenData*	«PenData»	10
string	*Riff*	«RiffAudio»	11
string	*WaveAudio*	«WaveAudio*	12
string	*UnicodeText*	«UnicodeText»	13
string	*EnhancedMetafile*	«EnhancedMetafile»	14
string	*FileDrop*	«FileDrop»	15
string	*Locale*	«Locale*	16

Например, выражение:

```
DataFormats.Text
```

возвращает строку «Text», а выражение:

```
DataFormats.Dib
```

- строку «DeviceIndependentBitmap». Я буду описывать эти форматы их текстовыми именами, так как текстовые версии более точно отражают способ идентификации формата данных буфером обмена.

Форматы «Text» и «OEMText» являются 8-битными кодировками. «Text» — это кодировка ANSI, используемая большинством Windows-программ. Кодировка «OEMText» — набор символов, используемый MS-DOS-программами, работающими в текстовом режиме. Формат «OEMText» позволяет копировать, вырезать и вставлять данные окна командной строки MS-DOS. Элемент буфера обмена, идентифицируемый форматом «Locale», — это число, обычно соответствующее 8-битной кодировке символов и служащее для определения набора символов. Формат «UnicodeText» идентифицирует текст Unicode.

Строка «SymbolicLink» идентифицирует текст в формате SYLK (Symbolic Link), созданный Microsoft для программы работы с электронными таблицами MultiPlan. Строка «DataInterchangeFormat» идентифицирует текст в формате DIF (Data Interchange Format, формат обмена данными), созданном Software Arts для работы с электронными таблицами VisiCalc. Оба этих формата буфера обмена появились в Windows 1.0 и, как вы можете догадаться, сейчас используются редко,

Когда программа для Windows Forms указывает в методе *GetData* формат буфера обмена «Text», «OEMText», «UnicodeText», «SymbolicLink» или «DataInterchangeFormat», возвращается объект типа *String*.

Строки «Bitmap» и «DeviceIndependentBitmap» идентифицируют соответственно аппаратно зависимые и аппаратно независимые растровые изображения. Формат «Palette» идентифицирует цветовую палитру, используемую совместно с 256-цветовыми DIB (аппаратно независимыми растровыми изображениями). Строка «TaggedImageFileFormat» идентифицирует растровые изображения в формате TIFF.

Строки «MetaFilePict» и «EnhancedMetafile» означают соответственно старый и улучшенный формат метафайлов.

Когда программа для Windows Forms указывает в методе *GetData* формат буфера обмена «Bitmap» или «DeviceIndepedentBitmap», возвращается объект типа *Bitmap*. Для строки «Enhanced Metafile», *GetData* возвращает объект типа *Metafile*. Однако для строки «MetaFilePict» *GetData* возвращает объект типа *MemoryStream*. Это просто блок памяти, к которому программа может обращаться через методы *ReadByte* и *Read* класса *MemoryStream*. *GetData* возвращает для «MetaFilePict» объект *MemoryStream,* так как метафайлы старого стиля не хранятся в самом буфере обмена. Вместо этого в нем хранится небольшая структура языка C, ссылающаяся на описатель метафайла.

Строка «PenData» используется расширениями Windows для работы с пером (ныне забытыми). «RiffAudio»идентифицирует мультимедийные данные в формате RIFF (Resource Interchange File Format, формат файлов для обмена ресурсами), а «WaveAudio» идентифицирует звуковые файлы. В первой версии Windows Forms не было поддержки мультимедиа. Для этих форматов буфера обмена, а также для «Palette», *GetData* возвращает объект типа *MemoryStream*.

И, наконец, «FileDrop» идентифицирует список файлов, возможно, первоначально полученный из Windows Explorer. (Чтобы поместить в буфер объект «FileDrop», выберите один/несколько файлов, затем выполните Copy или Cut.) В программах для Windows Forms элемент «FileDrop» — это массив строк. Хотя можно использовать элементы «FileDrop» для работы с буфером обмена, этот формат гораздо чаще применяют для операций буксировки, которые я рассмотрю ниже.

Пока что большинство из этих дополнительных форматов не очень полезно для программиста Windows Forms. Когда Windows Forms помещает объект *String* в буфер обмена, объект *String* автоматически преобразовывается в форматы «Text» и «UnicodeText» для использования в других приложениях. Если другое приложение помещает текст (любого типа) в буфер, этот текст автоматически преобразовывается в объект *String* для программы, использующей Windows Forms.

Аналогично, как мы видели, программы для Windows Forms могут обмениваться растровыми изображениями и метафайлами, не применяя преобразования, необходимые в программах, не использующих Windows Forms,

Пять дополнительных статических полей, определенных в классе *DataFormats*, чаще применяют программы для Windows Forms:

Статические поля *DataFormats* (выборочно)

Тип	Поле	Значение
string	*CommaSeparatedValue*	«Csv»
string	*Html*	«HTML Format»
string	*Rtf*	«Rich Text Format»
string	*Serializable*	«WindowsForms10PersistentObject»
string	*StringFormat*	«System.String»

Эти форматы не определены в интерфейсе буфера обмена Win32, но первые три используются программами для Win32.

«Csv» — это текстовый формат, используемый программами, работающими с электронными таблицами и базами данных, для обмена данными, В блоке электронной таблицы, состоящем из строк и столбцов, столбцы разделяются запятыми, а строки — символами конца строки. Если «Csv» используется при работе с базами данных, поля разделяются запятыми, а записи — символами конца строки. Числа хранятся в удобочитаемом формате ASCII; текст заключается в кавычки.

Хотя «Csv» является текстовым форматом, вызов *GetData с* параметром «Csv» возвращает не объект типа *string,* а *MemoryStream,* содержащий текст с завершающим нулем. Извлечь объект *string* из объекта *MemoryStream* позволяет код:

```
rtemoryStream memorystream = (MemoryStream)data.GetData("Csv");
StreamReader streamreader = new StreamReader(memorystream);
string str = streamreader.ReadToEnd();
```

Кроме того, нужен оператор *using* для доступа к пространству имен *System.IO.* (Подробнее о классах *MemoryStream* и *StreamReader* см. приложение А.) Объект *string,* вероятно, будет заканчиваться символом '\x0000'.

Когда программа для Windows Forms указывает «HTML Format» или «Rich Text Format», метод *GetData* возвращает объект типа *string.* Однако в первом случае он будет содержать вместе с текстом тэги HTML, а во втором — может содержать или не содержать тэги RTF. Так как простой текст является подмножеством RTF, объект *string* может содержать и только простой текст,

Формат «WindowsFormslOPersistentObject» появляется, когда программа для Windows Forms копирует в буфер обмена объект *Bitmap* или *Metafile.* Тип объекта, возвращаемого *GetData,* — или *Bitmap,* или *Metafile.* Обычно этот формат не требуется.

Как я упоминал ранее, для формата «System.String» *GetData* возвращает объект типа *String.* Это полностью равносильно применению формата *typeof(String).*

Меню Paste этой программы содержит подпункты, позволяющие вставлять простой текст, RTF, HTML или CSV:

RichTextPaste.cs

```
//-----------------------------------------
// flichTextPaste.es © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
```

см. след. стр.

```
using System.IO;
using System.Windows.Forms;

class RichTextPaste:  Form
{
     string   strPastedText = "";
     MenuItem miPastePlain, miPasteRTF, miPasteHTHL, miPasteCSV;

     public static void Main()
     {
          Application.Run(new RichTextPaste());
     }
     public RichTextPaste()
     {
          Text = "Rich-Text Paste";
          ResizeRedraw = true;

          Menu = new MainMenu();

               // Меню Edit

          MenuItem mi * new MenuItem("&Edit");
          mi.Popup += new EventHandler(MenuEditOnPopup);
          Menu.MenuItems.Add(mi);

               // Подпункт Paste Plain Text меню Edit

          miPastePlain = new MenuItem("Paste &Plain Text");
          miPastePlain.Click += new EventHandler(MenuEditPastePlainOnClick);
          Menu.MenuItems[0].MenuItems.Add(miPastePlain);

               // Подпункт Paste RTF меню Edit

          miPasteRTF = new MenuItem("Paste &Rich Text Format");
          ffllPaateRTF.Click += new EventHandler(MenuEditPasteRTFOnClick);
          Menu.MenuItems[0].MenuItems.Add(miPasteRTF);

               // Подпункт Paste HTML мен» Edit

          miPasteHTHL = new MenuItem("Paste &HTML");
          miPasteHTML.Click += new EventHandler(MenuEditPasteHTMLOnClick);
          Menu.MenuItems[0].MenuItems.Add(miPasteHTML);

               // Подпункт Paste CSV меню Edit

          miPasteCSV = new MenuItem("Paste &Comma-Separated Values");
          miPasteCSV.Click += new EventHandler(MenuEditPasteCSVOnClick);
          Menu.MenuItems[0].MenuItems.Add(miPasteCSV);
     }
     void MenuEditOnPopup(object obj, EventArgs ea)
```

см. след. стр.

```
{
    miPastePlain.Enabled =
        Clipboard.GetDataObject().GetDataPresent(typeof(string));
    miPasteRTF.Enabled -
        Clipboard.GetDataObject().GetDataPresent(DataFormats.Rtf);
    miPasteHTML.Enabled =
        Clipboard.GetDataObject().GetDataPresent(DataFormats.Html);
    miPasteCSV.Enabled =
        Clipboard.GetDataObject().GetDataPresent
                                (DataFormats.CommaSeparatedValue);
}
void MenuEditPastePlainOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    if (data.GetDataPresent(typeof(string)))
    {
        strPastedText = (string) data.GetData(typeof(string));
        Invalidate();
    }
}
void MenuEditPasteRTFOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    if (data.GetDataPresent(DataFormats.fitf ))
    {
        strPastedText = (string) data.GetData(DataFormats.Rtf); '
        Invalidate();
    }
}
void MenuEditPasteHTMLOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    if (data.GetDataPresent(DataFormats.Html))
    {
        strPastedText - (string) data.GetData(DataFormats.Html);
        Invalidate();
    }
}
void MenuEditPasteCSVOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    if (data.GetDataPresent(DataFormats.CommaSeparatedValue))
    {
        MemoryStream memstr = (MemoryStream) tfata.GetDataC'Csv");
        StreamReader streamreader - new StreamReader(memstr);
        strPastedText = streamreader.ReadToEnd();
```

см. след. стр.

```
                Invalidate();
            }
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics grfx = pea.Graphics;

            grfx.DrawString(strPastedText, Font, new SolidBrush(ForeColor),
                            ClientRectangle);
        }
}
```

Попробуйте выбрать какой-либо текст в Web-браузере, скопировать его в буфер обмена и затем с помощью этой программы посмотреть доступные форматы. Для формата CSV код преобразования объекта *MemoryStream* в *string* аналогичен коду, который я показывал ранее. Для форматов RTF и HTML программа просто выводит текст, не пытаясь анализировать тэги форматирования.

Программы просмотра буфера обмена

Программа просмотра буфера обмена (clipboard viewer) выводит текущее содержимое буфера обмена. Следующая программа просмотра буфера обмена содержит 21 переключатель, соответствующий 21 полю класса *DataFormats*. Она устанавливает 1-секундный таймер и проверяет содержимое буфера обмена по событию *Tick*. (Сообщения Win32, информирующие приложение, что содержимое буфера изменилось, напрямую недоступны программам для Windows Forms.) Переключатели активизируются в соответствии с доступными форматами. При щелчке кнопки элемент буфера обмена этого формата показывается в правой части формы.

ClipView.cs

```
//-------------------------------------
// ClipView.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;

class ClipView: Fora
{
    string[] astrFormats =
    {
    DataFormats.Bitmap, DataFormats.CommaSeparatedValue, DataFormats.Dib,
    DataFormats.Dif, DataFormats.EnhancedMetafile, DataFormats.FileDrop,
    DataFormats.Html, DataFormats.Locale, DataFormats.MetafilePict,
    DataFormats.OemText, DataFormats.Palette, DataFormats.PenData,
    DataFormats.Riff, DataFormats.Rtf, DataFormats.Serializable,
```

см. след. стр.

```
DataFormats.StringFormat, DataFormats.SymbolicLink, DataFormats.Text,
DataFormats.Tiff, DataFormats.UnicodeText, DataFormats.WaveAudio
};

Panel         panelDisplay;
RadioButton[] aradio;
RadioButton   radioChecked;

public static void Main()
{
    Application.flun(newClipView());
}
public  ClipViewO
i
    Text = "Clipboard Viewer";

        // Создаем панель переменной ширины для показа буфера обмена.

    panelDisplay = new Panel();
    panelDisplay.Parent = this;
    panelDisplay.Dock = DockStyle.Fill;
    panelDisplay.Paint += new PaintEventHandler(PanelOnPaint);
    panelDisplay.BorderStyle = BorderStyle.Fixed3D;

        // Создаем разделитель.

    Splitter split = new Splitter();
    split.Parent * this;
    split.Dock = DockStyle.Left;

        // Создаем панель для переключателей.

    Panel panel = new Panel();
    panel.Parent = this;
    panel.Dock = DockStyle.Left;
    panel.Width = 200;

        // Создаем переключатели.

    aradio = newfladio8titton[astrForfflats.LengttiJ;
    EventHandler eh = new EventHandler(RadioButtonOnClick);

    for (int i = 0; i < astrFormats.Length; i++)
    {
        aradio[i] - new RadioButton();
        aradio[i].Parent = panel;
        aradio[i].Location = new Point(4, 12 * i);
        aradio[i].Size = new Size(300, 12);
        aradio[i].Click += eh;
        aradio[i].Tag = astrFormats[i];
```

См. след. стр.

```
        }
            // Определяем базовый размер для автоматического масштабирования.

        AutoScaleBaseSize = new Size(4, 8);

            // Устанавливаем таймер на 1 секунду.

        Timer timer = new Timer();
        timer.Interval = 1000;
        timer.Tick += new EventHandler(TimerOnTick);
        timer.Enabled = true;
    }
    void TimerOnTick(object obj, EventArgs ea)
    {
        IDataObject data = Clipboard.GetDataObject();

        for (int i = 0; i < astrFormats.Length; i++)
        {
            aradio[i].Text = astrFormats[i];
            aradio[i].Enabled = data.GetDataPresent(astrFormats[i]);

            if (aradio[i].Enabled)
            {
                if (!data.GetDataPresent(astrFormats[i], false))
                    aradio[i].Text += "*";

                object objClip = data.GetData(astrFormats[i]);

                try
                {
                    aradio[i].Text += " (" + objClip.GetType() + ")";
                }
                catch
                {
                    aradio[i].Text += " (Exception on GetType!)";
                }
            }
        }
        panelDisplay.Invalidate();
    }
    void RadioButtonOnClick(object obj, EventArgs ea)
    {
        radioChecked = (RadioButton) obj;
        panelDisplay.Invalidate();
    }
    void PanelOnPaint(object obj, PaintEventArgs pea)
    {
        Panel    panel = (Panel) obj;
        Graphics grfx  = pea.Graphics;
        Brush    brush = new SolidBrush(panel.ForeColor);
```

см. след. стр.

```
            if (radioChecked == null || !radioChecked.Enabled)
                return;

            IDataObject data = Clipboard.GetDataObject();

            object objClip = data.GetData((string) radioChecked.Tag);

            if (objClip == null)
                return;

            else if (objClip.GetType() == typeof(string))
            {
                grfx.DrawString((string)objClip, Font, brush,
                                panel.ClientRectangle);
            }
            else if (objClip.GetType() == typeof(string[]))   // FileDrop
            {
                string str = string.Join("\r\n", (string[]) objClip);

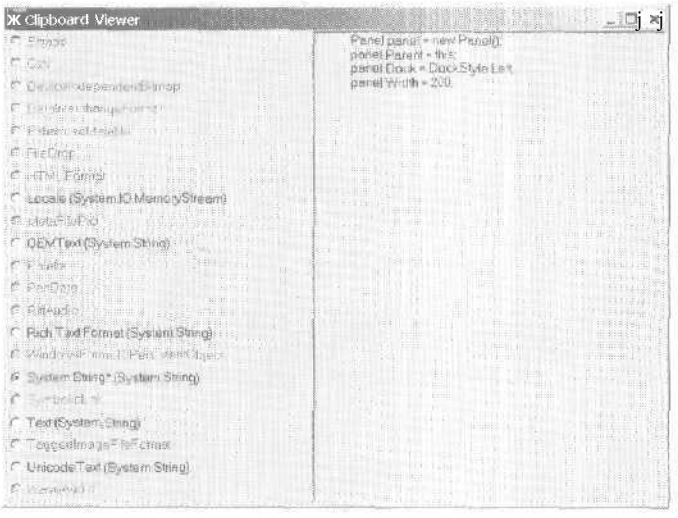
                grfx.DrawString(str, Font, brush, panel.ClientRectangle);
            }
            else if (objClip.GetType() == typeof(Bitmap) ||
                     objClip.GetType() == typeof(Metafile) ||
                     objClip.GetType() == typeof(Image))
            {
                grfx.DrawImage((Image)objClip, 0, 0);
            }
            else if (objClip.GetType() == typeof(MemoryStream))
            {
                Stream stream = (Stream) objClip;
                byte[] abyBuffer = new byte[16];
                long   lAddress = 0;
                int    iCount;
                Font   font = new Font(FontFamily.GenericMonospace,
                                       Font.SizeInPoints);
                float  y = 0;

                while ((iCount = stream.Read(abyBuffer, 0, 16)) > 0)
                {
                    string str = HexDump.ComposeLine(lAddress, abyBuffer,
                                                     iCount);
                    grfx.DrawString(str, font, brush, 0, y);
                    lAddress += 16;

                    y += font.GetHeight(grfx);
                    if (y > panel.Bottom)
                        break;
                }
            }
        }
    }
```

В обработчике события *Tick* тексту кнопки присваивается текстовое обозначение формата буфера обмена. Если формат доступен и получается преобразованием из другого формата буфера обмена, в текст кнопки добавляется звездочка, В скобках указываются типы данных .NET форматов буфера обмена.

Вот как выглядела программа, после того как я скопировал часть текста программы из Visual O .NET в буфер обмена:



Метод *PanelOnPaint* обновляет панель справа. Он может работать с несколькими типами данных .NET. Для типа *string* методом *DrawText* просто выводится текст. *DrawText* также служит для показа массива объектов *string,* используемого для формата буфера обмена «FileDrop». Для типов данных *Bitmap* или *Metafile,* метод *PanelOnPaint* вызывает *DrawImage.* А для типа данных *MemoryStream* он использует статический метод *ComposeLine* программы HexDump приложения А.

В программе ClipView перечисляются не все форматы данных буфера обмена. В ней показываются только те форматы, которые напрямую поддерживаются Windows Forms посредством полей класса *DataFormats.* Как я уже говорил, метод *GetFormats* интерфейса *IDataObject* позволяет получить массив объектов *string,* в котором перечислены все форматы текущего содержимого буфера обмена. Эту возможность я и использую в программе ClipViewAll.

ClipViewAtl.cs

```
//---------------------------------------
// ClipViewAll.cs fl 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;
```

см. след. стр.

```
class ClipViewAll: Form
{
     Panel      panelDisplay, panelButtons;
     RadioButton radioChecked;
     string[]    astrFormatsSave = new string[0];

     public static void Main()
     {
          Application.Run(new ClipViewAll());
     }
     public ClipViewAll()
     {
          Text = "Clipboard Viewer (All Formats)";

               // Создаем панель переменной ширины для показа буфера обмена.

          panelDisplay = new Panel();
          panelDisplay.Parent = this;
          panelDisplay.Dock = DockStyle.Fill;
          panelDisplay.Paint += new PaintEventHandler(PanelOnPaint);
          panelDisplay.BorderStyle = BorderStyle.Fixed3D;

               // Создаем разделитель.

          Splitter split = new Splitter();
          split.Parent = this;
          split.Dock = DockStyle.Left;

               // Создаем панель для переключателей,

          panelButtons = new Panel();
          panelButtons.Parent = this;
          panelButtons.Dock = DockStyle.Left;
          panelButtons.AutoScroll = true;
          panelButtons.Width = Width / 2;

               // Устанавливаем таймер на 1 секунду.

          Timer timer = new Timer();
          timer.Interval = 1000;
          timer.Tick += new EventHandler(TimerOnTick);
          timer.Enabled = true;
     }
     void TimerOnTick(object obj, EventArgs ea)
     {
          IDataObject data = Clipboard.GetDataObject();

          string[] astrFormats = data.GetFormats();
          bool bUpdate = false;
```

См. след. стр.

```
        // Определяем, изменились ли форматы буфера обмена.

    if (astrFormats.Length != astrFormatsSave.Length)
        bUpdate = true;
    else
    {
        for (int i = 0; i < astrFormats.Length; i++)
            if (astrFormats[i] != astrFormatsSave[i])
            {
                bUpdate = true;
                break;
            }
    }
        // В любом случае объявляем panelDisplay недействительной.

    panelDisplay.Invalidate();

        // Не изменяем переключатели, если форматы не изменились.

    if (!bUpdate)
        return;

        // Форматы изменились, поэтому заново создаем переключатели.

    astrFormatsSave = astrFormats;
    panelButtons.Controls.Clear();
    Graphics grfx = CreateGraphics();
    EventHandler eh = new EventHandler(RadioButtonOnClick);
    int cxText = AutoScaleBaseSize.Width;
    int cyText = AutoScaleBaseSize.Height;

    for (int i = 0; i < astrFormats.Length; i++)
    {
        RadioButton radio = new RadioButton();
        radio.Parent = panelButtons;
        radio.Text = astrFormats[i];

        if (!data.GetDataPresent(astrFormats[i], false))
            radio.Text += "*";

        try
        {
            object objClip = data.GetData(astrFormats[i]);
            radio.Text += " (" + objClip.GetType() + ")";
        }
        catch
        {
            radio.Text += " (Exception on GetData or GetType!)";
        }
        radio.Tag = astrFormats[i];
```

см. след. стр.

```
            radio.Location = new Point(cxText, i * 3 * cyText / 2);
            radio.Size = new Size((radio.Text.Length + 20) * cxText,
                                  3 * cyText / 2);
            radio.Click += eh;
        }
        grfx.Dispose();
        radioChecked = Roll;
    }
    void RadioButtonOnClick(object obj, EventArgs ea)
    {
        radioChecked = (RadioButton) obj;
        panelDisplay.Invalidate();
    }
    void PanelOnPaint(object obj, PaintEventArgs pea)
    {
        Panel    panel = (Panel) obj;
        Graphics grfx  = pea.Graphics;
        Brush    brush = new SolidBrush(panel.ForeColor);

        if (radioChecked == null)
            return;

        IDataObject data = Clipboard.GetDataObject();
        object objClip = data.GetData((string) radioChecked.Tag);

        if (objClip == null)
            return;

        else if (objClip.GetType() == typeof(string))
        {
            grfx.DrawString((string)objClip, Font, brush,
                            panel.ClientRectangle);
        }
        else if (objClip.GetType() == typeof(string[]))   // FileDrop
        {
            string str = string.Join("\r\n", (string[]) objClip);

            grfx.DrawString(str, Font, brush, panel.ClientRectangle);
        f
        else if (objClip.GetType() == typeof(Bitmap) j|
                 objClip.GetType() == typeof(Metafile) ||
                 ObjClip.GetTypeO == typeof(Image))
        {
            grfx.DrawImage((Image)objClip, 0, 0);
        }
        else if (objClip.GetType() == typeof(MemoryStream))
        {
            Stream stream = (Stream) objClip;
            byte[] abyBuffer = new byte[16];
            long   lAddress = 0;
```

См. след. стр.

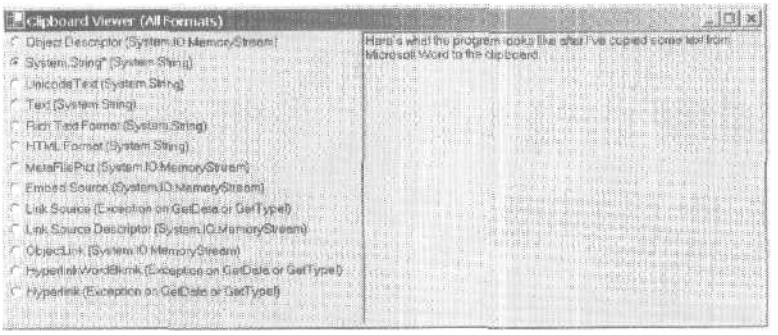
```
                int    iCount;
                Font   font = new Font(FontFamily.GenericMonospace,
                                       Font.SizeInPoints);
                float  y = 0;

                while ((iCount = stream.Read(abyBuffer, 0, 16)) > 0)
                {
                    string str = HexDump.ComposeLine(lAddress, abyBuffer,
                                                     iCount);
                    grfx.DrawString(str, font, brush, 0, y);
                    lAddress += 16;
                    y += font.GetHeight(grfx);

                    if (y > panel.Bottom)
                        break;
                }
            }
        }
    }
```

ClipViewAll ежесекундно проверяет, изменились ли форматы буфера обмена, и. если да, заново создает набор переключателей — по одному для каждого формата. Как и в ClipView, текст каждого из переключателей показывает также, является ли формат «родным», и соответствующий ему тип данных .NET.

Вот какой вид приобрела программа, после того как я скопировал текст из Microsoft Word в буфер обмена:



Да, здесь действительно видно, что буфер обмена содержит 13 различных форматов для одного фрагмента текста.

Так как буфер обмена обеспечивает промежуточный уровень обмена данными между приложениями, нужно проверять, как код работы с буфером обмена взаимодействует с другими приложениями. Программы ClipView и ClipViewAll могут пригодиться при исследовании буфера обмена с точки зрения программы для Windows Forms, но данные, копируемые вашей программой в буфер обмена, должны иметь смысл и для программ, не использующих Windows Forms,

Если начать изучать форматы буфера обмена, применяемые приложениями (например, список, показываемый ClipViewAll для Word), обнаружится, что некоторые из этих форматов используют OLE (object linking and embedding, связывание

и внедрение объектов), тогда как другие форматы (такие как «HyperlinkWordBknk»), очевидно, являются собственными форматами приложения.

В использовании собственного формата буфера обмена ничего сложного. Нужно просто создать имя для этого формата, записать его в *string u* задействовать этот объект *string* в качестве формата буфера обмена. Чтобы избежать конфликтов с другими приложениями, также использующими собственные форматы, нужно давать форматам гарантированно уникальные имена. Этого легко добиться, задавая имя приложения как составную часть имени формата буфера обмена, например «WriteALot Version 2.1 Formatted Text».

Win32-программы обычно обращаются к форматам буфера обмена по идентификационным номерам. (Первая таблица полей *DataFormats*, показанная выше, содержит идентификационные номера, соответствующие стандартным форматам буфера обмена.) Кроме того, Windows присваивает идентификационные номера нестандартным форматам, используемым приложениями. У класса *DataFormats* есть статический метод *GetFormat*, преобразующий имя формата и идентификационный номер формата:

Статические методы *DataFormats*

```
Format  GetFormat(int  id)
Format GetFormat(string strName)
```

Format — это еще один класс, определенный в *DataFormats,* соответственно в документации по библиотеке классов он описывается как *DataFormats.Format*. У класса всего два свойства только для чтения:

Свойства *Format*

Тип	Свойство	Доступ
int	*Id*	Чтение
string	*Name*	Чтение

Например, вызов:

```
DataFormats.GetFormat("DeviceIndependentBitmap").Id
```

возвратит 8, а:

```
DataFormats.GetFormat(8).Name
```

— строку «DeviceIndependentBitmap». Для нестандартных форматов возвращаемые числа изменяются от сеанса к сеансу, поэтому избегайте жесткого кодирования номеров нестандартных форматов.

Если вы используете собственный формат буфера обмена, дополните его стандартными форматами. Значит, требуется использовать для одного и того же элемента буфера обмена несколько форматов.

Использование нескольких форматов буфера обмена

Вы видели, что данные в буфер обмена помещает статический метод *SetDataObject* класса *Clipboard.* Например, текст в буфер обмена помещается так:

```
Clipboard.SetDataObject(strText);
```

Аналогичные вызовы используются, чтобы помесгить в буфер обмена объекты типа *Bitmap* или *Metafile,*

А если переменная *strText* содержит блок текста RTF или HTML? Как отразить этот факт при вызове *SetDataObject?* И что делать, если нужно поместить в буфер несколько форматов? Так как каждый вызов *SetDataObject* замещает содержимое буфера обмена, это кажется невозможным.

Решить эти две проблемы позволяет класс *DataObject.* Вспомните, что статический метод *GetDataObject* класса *Clipboard* описывается в документации как возвращающий объект класса, реализующего интерфейс *IDataObject. DataObject* — единственный класс .NET Framework, реализующий интерфейс *IDataObject.* (Это не значит, что метод *Clipboard.GetDataObject* обязательно должен возвращать объект типа *DataObject.* Он может динамически создавать новый класс. Но вы увидите, что *ClipboardGetDataObject* во многих случаях возвращает объект типа *DataObject.*)

У *DataObject* три конструктора:

Конструкторы *DataObject*

```
DataObject()
DataObject(object objData)
DataObject(string strFormat, object objData)
```

Например, вызов:

```
Clipboard.SetDataObject(strText);
```

равносилен:

```
Clipboard.SetDataObject(new DataObject(strText));
```

или:

```
Clipboard.SetDataObject(new DataObject(DataFormats.StringFormat, strText));
```

Если *strText* в действительности содержит блок HTML, можно использовать:

```
Clipboard.SetDataObject(new DataObject(DataFormats.Html, strText));
```

если текст не следует оставлять в буфере обмена после завершения программы, и:

```
Clipboard.SetDataOb|ect(new DataObject(DataFormats.Html, strText), true);
```

если текст нужно оставить.

DataObject реализует интерфейс *IDataObject,* так что он поддерживает все методы, определенные в *IDataObject;* других методов или свойств этот класс не поддерживает. Я уже рассматривал методы *GetDataPresent, GetData* и *GetFormats.* Осталось рассмотреть метод *SetData,* который и позволяет применить *DataObject* для определения нескольких форматов:

Метод *SetData* класса *DataObject*

```
void SetData(object objData)
void SetData(Type typeFormat, object objData)
void SetData(string strFormat, object objData)
void SetData(string strFormat, bool bConvert, object objData)
```

Сначала создается объект типа *DataObject,* затем при помощи *SetData* в него записывается несколько форматов одного элемента, и затем объект *DataObject* передается методу *Clipboard.SetDataObject.* По умолчанию элементы преобразуются в совместимые форматы. Исключение — случай, когда вызывается последняя перегруженная версия, показанная в этой таблице, и параметр *bConvert* имеет значение *false.*

Допустим, что *strText* — строка простого текста, которую вы хотите скопировать в буфер обмена, *strHtml* содержит тот же текст с форматированием HTML, а *strRtf* — та же строка с форматированием RTF. Вот последовательность операторов, записывающая эти несколько форматов в буфер обмена:

```
DataObject data = new DataObject();
data.SetData(strText);
data.SetData(DataFormats.Html, strHtml);
data.SetData(DataFormats.Rtf, strRtf);
Clipboard.SetDataObject(data, true);
```

Будьте внимательны: никто не проверяет, действительно ли *strHtml* и *strRtf* являются блоками HTML и RTF! Аналогично любые данные, получаемые программой из буфера обмена, могут не соответствовать своему формату. Требуется очень тщательно анализировать строки HTML и RTF. получаемые из буфера обмена.

Программа Multicopy копирует в буфер обмена двумерный массив значений типа *float* (хранящийся в поле *afValues*) в трех различных форматах: собственном формате, формате CSV и в формате простого текста:

MultiCopy.cs

```
// ------------------ ..... ----- ..... -
// MultiCopy.cs @ 2081 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class MultiCopy: Form
{
     const string strFormat = "MultiCopy.InternalFormat";

     float[,] afValues = {{ 0.12f, 3.45f, 6.78f, 9.01f },
                          { 2.34f, 5.67f, 8.90f, 1.23f },
                          { 4.56f, 7.89f, 0.12f, 3.45f }};
```

см. след. стр.

```
public static void Main()
{
     Application.Run(new MultiCopyO);
}
public  MultiCopyO
{
     Text  =  "Multi  Copy";
     Menu * new MainMenu();

          // Кенв Edit

     MenuItem mi - new MenuItem("&Edit");
     Menu.MenuItems.Add(mi);

          // Пункт Copy меню Edit

     mi = new MenuItem("&Copy");
     mi.Click += new EventHandler(MenuEditCopyOnClick);
     mi.Shortcut = Shortcut.CtrlC;
     Menu.MenuItems[0].MenuItems.Add(mi);
}
void MenuEditCopyOnClick(object obj, EventArgs ea)
i
     DataObject data = new DataObject();

          // Определяем свой внутренний формат буфера обмена.

     MemoryStream memorystream = new MemoryStream();
     BinaryWriter binarywriter = new BinaryWriter(memorystream);

     binarywriter.Write(afValues.GetLength(0));
     binarywriter.Write(afValues.GetLength(1));

     for (int iRow = 0; iRow < afValues.GetLength(0); iRow++)
     for (int iCol = 0: iCol < afValues.GetLength(1); iCol++)
          binarywriter.Write(afValues[iRow, iCol]);

     binarywriter.Close();

     data.SetData(strFormat, memorystream);

          // Определяем формат CSV буфера обмена.

     memorystream = new MemoryStream();
     StreamWriter streamwriter = new StreamWriter(memorystream);

     for  (int iRow * 0;  iRow < afValues.GetLength(0); iRow++)
     for (int iCol = 0;  iCol < afValues.GetLength(1); iCol++)
     i
          streamwriter.Write(afValues[iRow, iCol]);
```

см. след. стр.

```
            if (iCol < afValues.GetLength(1) - 1)
                streamwriter.Write(",");
            else
                streamwriter.WriteLine(); '.
        }
        streamwriter.Write("\0");
        streamwriter.Close();
        data.SetData(DataFormats.CommaSeparatedValue, memorystream);

            // Определяем формат String буфера обмена.

        StringWriter stringwriter = new StringWriter();

        for (int iRow = 0; iRow < afValues.GetLength(0); iRow++)
        for (int iCol - 0; iCol < afValues.GetLength(1); iCol++)
        {
            stringwriter.Write(afValues[iRow, iCol]);

            if (iCol < afValues.GetLength(1) - 1)
                stringwriter.Write("\t");
            else
                stringwriter.WriteLine();
        }
        stringwriter.Close();
        data.SetData(stringwriter.ToString());

        Clipboard.SetDataObject(data, false);
    }
}
```

Большую часть Multicopy составляет обработчик события *MenuEditCopyOnClick*. Этот метод начинается определением объекта типа *DataObject* и кончается вызовом *Clipboard.SetDataObject* для копирования объекта в буфер обмена. В промежутке между этими действиями данные форматируются тремя способами.

Собственный формат идентифицируется строкой «MultiCopy.InternalFormat». Массив значений типа *float* (перед которым идут целые количества строк и столбцов) записывается в двоичном формате в объект *MemoryStream*. В методе используется класс *BinaryWriter* (см. о нем приложение А) для записи в поток двоичных объектов. Поток данных в памяти (объект типа *MemoryStream)* добавляется в объект *DataObject* оператором:

```
data.SetData(strFormat, memorystream);
```

в котором *strFormat* — строка «MultiCopy.InternalFormat».

Затем метод форматирует данные как CSV. Опять создается объект *MemoryStream* для хранения данных, и на этот раз создается объект *StreamWriter* для добавления в поток форматированных текстовых строк. Значения в одной и той же строке массива разделяются запятыми; строки массива разделяются символами возврата каретки и перевода строки. Поток данных в памяти добавляется в объект *DataObject* оператором:

```
data.SetData(DataFormats.CommaSeparatedValue, memorystream);
```

И, наконец, данные форматируются как простой текст. Это похоже на форматирование CSV за исключением того, что значения одной строки массива разделяют символы табуляции, а не запятые. Еще одно отличие в том, что текст не помещается в поток данных в памяти — вместо этого с помощью объекта *StringWriter* строится строка, содержащая форматированный текст. Этот формат добавляется в объект *DataObject* оператором:

```
data.SetData(stringwriter.ToString());
```

Буксировка

Буксировка позволяет захватывать что-либо мышью и перетаскивать в новое место того же или другого приложения. Обычно пользователь захватывает один или несколько файлов или блок текста, хотя можно буксировать изображения и другие типы данных.

Приложение, из которого перетаскивается объект, называется *источником* буксировки (drag-and-drop source), а то, в которое он перетаскивается, — *адресатом* (drag-and-drop target). При буксировке обычно требуется держать нажатой левую кнопку стандартно настроенной мыши, хотя некоторые приложения допускают буксировку с нажатой правой кнопкой мыши.

Windows Explorer часто является источником буксировки списка файлов, которому соответствует тип данных буфера обмена «FileDrop». Например, если выбрать файл в Explorer и затем перетащить его в клиентскую область Notepad, то Notepad откроет и покажет файл. Программа Microsoft может быть как источником, так и адресатом буксировки текста. Текст, выделенный в WordPad, можно перетащить в другое приложение, способное выступать в качестве адресата буксировки. Аналогично и WordPad может быть адресатом буксируемого текста.

Данные, перетаскиваемые из одного приложения в другое, обычно копируются, переносятся или связываются. Это зависит от состояния клавиш Shift и Ctrl:

Действия при буксировке

Нажатая клавиша	Действие
Нет	Перенос (Move)
Ctrl	**Копирование** (Copy)
Shift+Ctrl	Связывание (Link)

При переносе объект удаляется из источника буксировки. При копировании адресат принимает копию объекта. При связывании источник и адресат получают ссылку на один объект.

Если вы захватите мышью список файлов Explorer или блок текста WordPad и перетащите его в клиентскую область любой из программ, которые вы пока что видели в этой книге, курсор примет вид перечеркнутого круга — международного знака, означающего запрет. Чтобы стать адресатом буксировки, элемент управления или форма должны присвоить *true* свойству *AllowDrop*:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
bool	*AllowDrop*	Чтение/запись

Следующие четыре события используются адресатами буксировки:

События *Control* (выборочно)

Событие	Метод	Делегат	Параметры
DragEnter	*OnDragEnter*	*DragEventHandler*	*DragEventArgs*
DragOver	*OnDragOver*	*DragEventHandler*	*DragEventArgs*
DragDrop	*OnDragDrop*	*DragEventHandler*	*DragEventArgs*
DragLeave	*OnDragLeave*	*EventHandler*	*EventArgs*

Событие *DragEnter* происходит, когда свойство *AllowDrop* элемента управления или формы имеет значение *true* и курсор мыши с перетаскиваемым объектом начинает попадать на элемент управления или клиентскую область формы. После события *DragEnter* при перемещении курсора на элементе управления или в клиентской области происходит событие *DragOver*. Если курсор затем выходит за пределы элемента управления или клиентской области, происходит событие *DragLeave*.

При возникновении события *DragEnter* или последующих событий *DragOver* элемент управления или форма могут сообщать о готовности стать адресатом перетаскиваемых данных. (Мы вскоре увидим, каким образом.) При этом курсор из перечеркнутого крута превращается в стрелку с прямоугольником на конце, возможно со знаком «плюс» (при копировании) или с изогнутой стрелкой (при связывании). Если затем отпустить кнопку мыши над клиентской областью, происходит событие *DragDrop*.

Если элемент управления или форма не сообщат о своей готовности принять данные, курсор по-прежнему имеет вид перечеркнутого крута. При этом, если отпустить кнопку мыши над клиентской областью, происходит событие *DragLeave*, а не *DragDrop*.

Обычно требуется обрабатывать события *DragOver* и *DragDrop*. При обработке события *DragOver* нужно решить, можно ли принять данные, перетаскиваемые на элемент управления или форму. Если отпустить кнопку позволяется только в определенных областях элемента управления или формы, можно сообщить, когда отпустить кнопку можно, а когда нет. При обработке события *DragDrop* осуществляется обращение к перетаскиваемым данным. (Это аналогично вставке из буфера обмена.)

i 1ри событиях *DragEnter, DragOver* и *DragDrop* передается объект *DragEventArgs* с такими свойствами:

Свойства *DragEventArgs*

Тип	Свойство	Доступ
in	*KeyState*	Чтение
im	*X*	Чтение
int	*Y*	Чтение
IDataObject	*Data*	Чтение

Свойства *DragEventArgs* (*продолжение*)

Тип	Свойство	Доступ
DragDropEffects	*AllowedEffect*	Чтение
DragDropEffects	*Effect*	Чтение/запись

Первые три свойства предоставляют информацию о состоянии клавиатуры и мыши в момент возникновения события. В свойстве *KeyState* используется набор битовых флагов, показывающих, какие кнопки мыши и управляющие клавиши нажаты в данный момент:

Битовые флаги *KeyState*

Клавиша или кнопка	Битовый флаг
Левая кнопка мыши	0x01
Правая кнопка мыши	0x02
Клавиша Shift	0x04
Клавиша Ctrl	0x08
Средняя кнопка мыши	0x10
Клавиша Alt	0x20

Свойства *X* и *Y* указывают положение курсора мыши в экранных координатах. (Для преобразования в координаты клиентской области служит метод *PointToClient* класса *Control.)*

Свойство *Data* является объектом, относящимся к классу, реализующему интерфейс *IDataObject*, также как и значение, возвращаемое методом *GetDataObject* класса *Clipboard,* При событиях *DragEnter* или *DragOver* методами *GetFormats* или *GetDataPresent* можно определить, относятся ли данные к типу, который способна обработать программа. При событии *DragDrop* можно получить копии данных методом *GetData.*

Свойство *AllowedEffect* содержит члены перечисления *DragDropEffects:*

Перечисление *DragDropEffects*

Член	Значение
None	0x00000000
Copy	0x00000001
Move	0x00000002
Link	0x00000004
Scroll	0x80000000
All	0x80000003

Свойство *AllowedEffect* устанавливается источником буксировки для определения возможностей, доступных адресату буксировки. Чаще всего источник буксировки присваивает *AllowedEffect* значение:

```
DragDropEffects.Copy | DragDropEffects.Move | DragDropEffects.Link
```

При событиях *DragEnter* и *DragOver,* основываясь на свойствах *KeyState,* X, *Y, Data* и *AllowedEffect* объекта *DragEventArgs,* потенциальный адресат буксировки определяет, может ли он принять перетаскиваемые данные. Если да, то присваи-

вает свойству *Effect* один из членов перечисления *DragDropEffects*содержащийся в свойстве *AllowedEffect*.Обычно адресат с помощью свойства *KeyState* определяет, можно ли присвоить свойству *Effect* член перечисления *Copy, Move* или *Link*. (Этим и определяется вид курсора.) Присваивание свойству *Effect* значения *DragDrop-Effects.None*сообщает о том, что адресат не может принять данные. После отпускания объекта пользователем источник буксировки информируется о том, какой член перечисления выбрал адресат.

При любой операции по буксировке потенциальному адресату буксировки достаточно присвоить значение свойству *Effect* только раз; это значение будет присутствовать в параметре *DragEventArgs*последующих событий *DragOver* и *Drag-Drop*. Это выглядит так, будто в течение всей операции буксировки используется один объект *DragEventArgs*. Однако обработчик события *DragOver* может изменить *Effect* в соответствии с текущим состоянием клавиш Ctrl и Shift,

Б главе 18 я начал серию программ, в которой пытался все более и более близко имитировать Notepad. Эта история продолжилась в главе 21, а сейчас я ее завершу. Следующий файл превращает программу в адресат буксировки текста или списка «FileDrop». (Настоящий Notepad является адресатом только для «FileDrop».) Я назвал его просто NotepadClone.cs, так как это последняя версия программы в данной книге.

NotepadClone.cs

```
//---------------------------------------
// NotepadClone.cs © 2001 fry Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class NotepadClone: NotepadCloneWithPrinting
{
     public new static void Main()
     {
          // Это нужно сделать, чтобы заработала буксировка.

          System.Threading.Thread.CurrentThread.ApartmentState =
                                        System.Threading.ApartmentState.STA;

          Application.Run(newttotepadClone0);
     }
     public NotepadClone()
     {
          strProgName = "NotepadClone";
          MakeCaption();

          txtbox.AllowDrop = true;
          txtbox.DragOver += new DragEventHandler(TextBoxOnDragOver);
          txtbox.DragDrop += new DragEventHandler(TextBoxOnDragDrop);
     }
```

см, след, стр.

```
    void TextBoxOnDragOver(object obj, DragEventArgs dea)
    {
        if (dea.Data.GetDataPresent(DataFormats.FileDrop) ||
            dea.Data.GetDataPresent(DataFormats.StringFormat))
        {
            if ((dea.AllowedEffect & DragDropEffects.Move) != 0)
                dea.Effect = DragDropEffects.Move;

            if (((dea.AllowedEffect & DragDropEffects.Copy) != 0) &&
                ((dea.KeyState & 0x08) != 0))    // Клавиша Ctrl
                dea.Effect = DragDropEffects.Copy;
        }
    }
    void TextBoxOnDragDrop(object obj, DragEventArgs dea)
    {
        if (dea.Data.GetDataPresent(DataFormats.FileDrop))
        {
            if (!OkToTrash())
                return;

            string[] astr = (string[])
                            dea.Data.GetData(DataFormats.FileDrop);

            LoadFile(astr[0]);      // В NotepadCloneWithFile.cs
        }
        else if (dea.Data.GetDataPresent(DataFormats.StringFormat))
        {
            txtbox.SelectedText =
                    (string) dea.Data.GetData(DataFormats.StringFormat);
        }
    }
}
```

В конструкторе свойству *AllowDrop* элемента управления *TextBox* присваивается *true*, и устанавливаются обработчики событий *DragOver* и *DragDrop* текстового поля. При событии *DragOver* проверяется наличие форматов данных «FileDrop» или «System.String», и свойству *Effect* объекта *DragEventArgs* присваивается или значение *DragDropEffects.Move* или значение *DragDropEffects.Copy* в зависимости от того, какие операции поддерживает источник буксировки и состояния клавиши Ctrl.

При событии *DragDrop* выполняются другие действия, зависящие от формата данных. При использовании формата «FileDrop» программа загружает файл. Хотя «FileDrop» обычно используется для *списка* файлов, NotepadClone может работать только с одним файлом. Для формата «System.String» выполняется операция, аналогичная Paste.

Вы можете поэкспериментировать с передачей текста в NotepadClone, используя эту программу с различными источниками буксировки, например, WordPad. При перетаскивании чего-либо из WordPad в NotepadClone, можно управлять видом курсора, нажимая и отпуская клавишу Ctrl. Когда наконец происходит событие *DragDrop*, WordPad уведомляется о последнем значении свойства *Effect*. Удалять или не удалять выделенный текст, определяет WordPad.

Теперь посмотрим, что значит быть источником буксировки. Любой класс, происходящий от *Control,* может инициировать операцию буксировки, вызывая такой метод (обычно при событии *MouseDown*):

Методы *Control* (выборочно)

```
DragDropEffects DoDragDrop(object objData, DragDropEffects dde)
```

Первый параметр — это объект, предлагаемый источником для буксировки. Этот параметр может быть объектом типа *DataObject*, если источник буксировки предоставляет данные в нескольких форматах или если нужно точнее определить формат данных (например, указать «HTML Format» для типа *string).* Второй параметр — один или несколько членов перечисления *DragDropEffects.*

Метод не возвращает управление, пока не завершится операция буксировки. В этот момент *DoDragDrop* возвращает член перечисления *DragDropEffects* указанный адресатом буксировки, или *DragDropEffects.None* если адресат не принял данные или операцию каким-либо образом отменили.

Хотя *DoDragDrop* не возвращает управление до завершения операции буксировки, элемент управления или форма могут периодически получать информацию о ходе процесса буксировки, обрабатывая событие *QueryContinueDrag.* Ниже многоточиями показано имя события в именах метода, делегата и параметра события:

События *Control* (выборочно)

Событие	Метод	Делегат	Параметр
QueryContinueDrag	*On...*	*...EventHandler*	*...EventArgs*

Объект *QueryContinueDragEventArgs* используемый событием, имеет такие свойства:

Свойства *QueryContinueDragEventArgs*

Тип	Свойство	Доступ
int	*KeyState*	Чтение
bool	*EscapePressed*	Чтение
DragAction	*Action*	Чтение/запись

Источник буксировки может присваивать свойству *Action* один из членов перечисления *DragAction:*

Перечисление *DragAction*

Член	Значение
Continue	0
Drop	1
Cancel	2

Обычно операция буксировки отменяется при нажатии пользователем клавиши Esc. Обрабатывая описанное выше событие, можно изменить это поведение или отменить буксировку по каким-либо другим причинам.

Эта программа, переопределяющая ImageClip и вносящая в нее усовершенствования, становится как источником, так и адресатом буксировки.

imageDrop.cs

```
//---------------------------------------
// ImageDrop.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;

class ImageDrop: ImageClip
{
     bool bIsTarget;

     public new static void Main()
     {
          Application.Run(new ImageDrop());
     }
     public ImageDrop()
     {
          Text = strProgName = "Image Drop";

          AllowDrop = true;
     }
     protected override void OnDragOver(DragEventArgs dea)
     {
          if (dea.Data.GetDataPresent(DataFormats.FileDrop) ||
              dea.Data.GetDataPresent(typeof(Metafile))      \\
              dea.Data.GetDataPresent(typeof(Bitmap)))
          {
               if ((dea.AllowedEffect & DragDropEffects.Move) != 0)
                    dea.Effect = DragDropEffects.Move;

               if (((dea.AllowedEffect & DragDropEffects.Copy) != 0) &&
                   ((dea.KeyState & 0x08) != 0))     // Клавиша Ctrl
                    dea.Effect = DragDropEffects.Copy;
          }
     }
     protected override void OnDragDrop(DragEventArgs dea)
     {
          if (dea.Data.GetDataPresent(DataFormats.FileDrop))
          {
               string[] astr = (stringE])
                                   dea.Data.GetData(DataFormats.FileDrop);

               try
               {
                    image = Image.FromFile(astr[0]);
               }
```

см. след. стр.

```
            catch (Exception exc)
            {
                MessageBox.Show(exc.Message, Text);
                return;
            }
            strFileName = astr[0];
            Text = strProgName + " - " + Path.GetFileName(strFileName);
            Invalidate();
        }
        else
        {
            if (dea.Data.GetDataPresent(typeof(Metafile)))
                Image = (Image) dea.Data.GetData(typeof(Metafile));

            else if (dea.Data.GetDataPresent(typeof(Bitmap)))
                image = (Image) dea.Data.GetData(typeof(Bitmap));

            bIsTarget = true;
            strFileName = "DragAndDrop";
            Text = strProgName + " - " + strFileName;
            Invalidate();
        }
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
        if (image != null)
        {
            blsTarget = false;

            DragDropEffects dde = DoDragDrop(image,
                        DragDropEffects.Copy | DragDropEffects.Move);

            if (dde == DragDropEffects.Move && !bIsTarget)
                image = null;
        }
    }
}
```

Методы *OnDragOver* и *OnDragDrop* аналогичны обработчикам событий *DragOver* и *DragDrop* программы NotepadClone. ImageDrop также становится источником буксировки, вызывая *DoDragDrop* в методе *OnMouseDown*. Программа разрешает выполнять копирование и перенос; если *DoDragDrop* возвращает *DragDropEffects.Move*, она действительно удаляет свою копию объекта *Image*, присваивая *null* переменной *image*.

Моя предыдущая версия этой программы работала не совсем правильно, когда я пытался выполнить операцию переноса внутри самой программы. Это происходило из-за того, что метод *DoDragDrop* возвращал управление после метода *OnDragDrop* и программа удаляла изображение, которое только что получила! Я завел переменную *blsTarget* исключительно для обработки этой ситуации: программа больше не удаляет изображение, переносимое внутри самой себя.

Предметныйуказатель

A

AboutBox 728
AgeOfInnocence 380
AllAboutFont 340
AnalogClock 416, 619, 621
AntiAlias 162
AntiAliasedText 359
API Win32 210
ArbitraryCoordinates 247
ARGB (alpha-red-green-blue) 99
ASCII 11, 190
AutoScaleDemo 592

B

BaselineTilt 902
BASIC 2, 8, 20
BetterBlockOut 285
BetterContextMenu 660
BetterDialog 726
BetterFamiliesList 352
BetterFontAndColorDialogs 750
BetterImageFromFile 433
BetterPieChart 181
Bezier 617
BezierArt 627
BezierCircles 625
BezierClock 621
BezierClockControl 619
BezierManual 631
BitmapButtons 562
BlockFont 894
BlockOut 282
BoldAndItalic 327
BoldAndItalicBigger 332
BoldAndItalicTighter 364
Bounce 474
BouncingGradientBrushBall 795
BoxingTheClient 164
Bricks 889
ButtonStyles 560

C

C* 2, 5
— hello-world 2, 6
— интерфейс 289
— исходный код 2, 3
— компилятор 3, 47
— компиляция 4
— обработка исключений 249
— перечисление 13
— пространство имен 6
— тип данных 9
——bool 11
— — byte 10, 11
— — char 11
— — decimal 10
— — double 10, 88
— -float 10, 88
— — int 8, 10
— — long 10
— — sbyte 10, 11
— — short 10
— — string 11
— — uint 10
— — ulong 10
— — ushort 10
— — псевдоним 43
— — размерный 12, 42, 83
— — с плавающей точкой 10, 88
— — ссылочный 12, 42, 83
— — структурный 83
— — числовой 515
— — целочисленный 10
CanonicalSpline 633
CanonicalSplineManual 639
CaptureLoss 287
CaptureLossNotifyWindow 290
Caret 216
CDate 21
CenterImage 440
CenterPixelSizeImage 442
CGA 139
CheckAndRadioCheck 655
CheckBoxDemo 579, 582, 586
CheckBoxWithLabel 583, 585
Checker 304
CheckerChild 308
CheckerChildWithFocus 311
CheckerWithChildren 309
CheckerWithChildrenAndFocus 312
CheckerWithKeyboard 306
CIL (Common Intermediate Language) 4
ClientEllipse 174
ClippingCombinations 708
ClipText 918, 1097
ClipView 1114
ClipViewAll 1118
ClockControl 412
ClosedCurveFillModes 637
CloseInFive 386
Clover 706.
CLR 13
CLR (Common Language Runtime) 4, 47
ColorFillDialogBox 731
ColorFillDialogBoxWithApply 736
ColorScroll 605, 609

ColorScrollDialogBox 740
ColorTrackBar 611
ConsoleAdder 8
ConsoleHelloWithUsing 6
ConsoleHelloWorld 2, 4, 5
ContextMenuAdd 663
ContextMenuDemo 658, 660. 663
CppDateClass 23
CppDateStruct 22
CreateMetafile 1072
CreateMetafileMemory 1076
CreateMetafileReload 1074
CsDateClass 24
CsDateConstructors 36
CsDateInheritance 38
CsDateProperties 32
CustomCheckBox 587

D
DashedEllipse 176
DateAndTimeStatus 937
DialogsWithRegistry 753
DIB (device-independent bitmap) 429
DigitalClock 404
DigitalClockWithDate 405
DirectoriesAndFiles 1034
Directory TreeView 1031
DLL (dynamic-link library) 3
DotsPerInch 230
DrawHouse 165
DrawOnImage 454
DrawOnPixelSizeImage 456
DrawOrFillEllipse 734
DrawOrFillEllipseWithApply 738
DropDownMenuButton 962
DropShadow 892
DropShadowWithPath 914
DualWink 473

E
EmbossedText 893
EnterLeave 295
EnumerateEnumeration 858
EnumerateEnumerationCombo 871
EnumMetafile 1088
EnvironmentVars 867
EvenBetterBlockOut 290
ExitOnX 193
ExplorerLike 1055

F
FamiliesList 353
FileListView 1052
FillModesClassical 185
FillModesOddity 186
FindReplaceDialog 839
FirstMainMenu 647
Flower 692
FontAndColorDialogs 744
FontMenu 665
FontMenuForm 888
FontNames 329
FontSizes 331
FormProperties 59, 60
FormSize 95
Fortran 20
FourByFours 183
FourCorners 110
FullFit 912

G
GDI (Graphics Device Interface) 63
GDI+ 234
GetFamiliesList 355
GradientPen 804
GradientText 890

H
HatchBrushArray 770
HatchBrushMenu 771
HatchBrushRenderingOrigin 778
HeadDump 756
Hello World 462
HELLO.C 44
HELLO.RC 44
HelloCenteredAlignment 111
HelloCenteredMeasured 113
HelloCenteredRectangle 114
HelloPrinter 157
hello-world 1
HelloWorld 75
HelloWorldBitmap 462
HelpMenu 678
HexagonGradientBrush 802
HexCalc 593, 594
HexDump 488
HitTestText 313
HollowFont 909
HollowFontCenteredPath 910
HollowFontWidened 917
HollowFontWidePen 915
HowdyWorld 341
HowdyWorldFullFit 343
HtmlDump 498
HuckleberryFinn 115
HuckleberryFinnHalfHeight 372
HundredPixelsSquare 227, 231

I
ImageAtPoints 447
ImageClip 1102
ImageDirectory 1041
ImageDrop 1134
ImageFromFile 432
ImageFromWeb 434
ImageIO 764
ImageOpen 761
ImagePanel 1037
ImagePrint 997
ImageReflection 445

ImageScaleIsotropic 444
ImageScaleToRectangle 443
IME (Input Method Editor) 199
Infinity 622
InheritHelloWorld 78
InheritTheForm 73, 74
InheritWithConstructor 74
InstalledFontsList 356
InstantiateHelloWorld 80
Intel 4
Intermediate Language Assembler 4
Intermediate Language Disassembler 4

J
Java *2,* 4, 5
JeuDeTaquin 420
JeuDeTaquinTile 418
JeuDeTaquinWithScramble 650

K
KeyExamine 205
KeyExamineWithScroll 211
KeyholeBitmap 712
KeyholeClip 710
KeyholeClipCentered 711

L
LineArcCombo 682
LineArcPath 684
LineCaps 814
LineJoins 810

M
MatrixElements 879
MatrixMethods 882
MeasureString 224
Menu Designer 642
MenuHelpFirstTry 939
MenuHelpSubclass 944
MenuItemHelp 943
MessageBoxHelloWorld 47
MetafileClip 1100
MetafileConvert 1069
MetafilePageUnits 1081
MetafileViewer 1064
MFC 686
MFC (Microsoft Foundation Class) 591
Microsoft Visual Studio .NET 45
MinAndMax 10
MobyDick 250
ModelessColorScroll 743
MouseConnect 279
MouseConnectWaitCursor 298
MouseCursors 298
MouseCursorsProperty 301
MouseWeb 277
MSIL 4
MSIL (Microsoft Intermediate Language) 4
Multicopy 1125
MyDocumentsFolder 51

N
NaiveFamiliesList 350
.NET
— DLL 4
— документирование 7
— исполняющая среда 3
— техническая документация 3
— тип данных 43
.NET Framework
— документация библиотек классов 41
— класс 42
— представление даты и времени 388
— соглашение об именовании 43
.NET Framework Software Development Kit (SDK) 3
NewForm 52, 53
Notepad 3
NotepadClone 1131
NotepadCloneNoMenu 824
NotepadCloneWithEdit 838, 845
NotepadCloneWithFile 830
NotepadCloneWithFormat 851
NotepadCloneWithPrinting 1002
NotepadCloneWithRegistry 826

O
OldFashionedMenu 651
OnePanelWithSplitter 1013
OutlineClientRectangle 172
OverlappingHatchBrushes 776
OwnerDrawButtons 565
OwnerDrawMenu 674

P
PaintEvent 64
PaintHello 69
PaintTwoForms 71
PartialImage 449
PartialImageRotate 452
PartialImageStretch 450
Pascal 2, 20
PathWarping 702
PenDashCaps 812
PenDashStyles 807
PenWidths 238
PictureBoxPlus 481
PictureBoxPlusDemo 482
PieChart 179
PInvoke (Platform Invocation Services) 210
PL/I 20
PoePoem 274
PolyEllipse 169
PostScript 624
PrintabieForm 159
PrintableTenCentimeterRuler 235
PrinterSelectionDialog 979
PrintThreePages 981
PrintWithMargins 986
PrintWithStatusBar 990
ProgramWithIcon 467

R

RadioButtons 599
RandomClear 106
RandomClearResizeRedraw 107
RandomRectangle 387
RectangleLinearGradientBrush 787
ReflectedText 897
RichTextPaste 1111
RotateAndReflect 899
RotatedFont 896
RotatedRectangles 264
RoundRect 177
RunFormBadly 56
RunFormBetter 57

S

Scribble 314
ScribbleWithBitmap 464
ScribbleWichPath 694
ScribbleWithSave 317
SDK 44
SeparateMain 77
SevenSegmentClock 410
SevenSegmentDisplay 407
ShowForm 54
ShowFormAndSleep 55
SimpleButton 550, 553
SimpleClock 402
SimpleDialog 717
SimplePrintDialog 993
SimplerDialog 722
SimpleShear 900
SimpleStatusBar 928
SimpleStatusBarWithPanel 932
SimpleToolBar 947
SimpleTreeView 1024
SineCurve 166
SixInchRuler 240
Spiral 170
SplitThreeAcross 1016
SplitThreeFrames 1018
SplitTwoProportional 1021
SquareTile 799
StandardMenu 669
StarGradientBrush 794
StatusBarAndAutoScroll 930
StatusBarPrincConcroller 989
StreamWriterDemo 496
StringAlignmentPoint 367
StringAlignmentRectangle 366
StringAppend 546
StringBuilderAppend 547
StringWriterAppend 548
SysInfoColumns 121
SysInfoEfficient 143
SysInfoFirstTry 119, 121
SysInfoKeyboard 202
SysInfoList 128
SysInfoListView 1048
SysInfoPanel 131, 135
SysInfoReflection 147
SysInfoReflectionStrings 144
SysInfoScroll 135, 136
SysInf o Strings 124
SysInfoUpdate 140
SysInfoUpdate.cs 140
SystemInformation 149

T

TallInTheCenter 922
TenCentimeterRuler 232, 235
TenCentimeterRulerAuto 236
TextBox 817
TextBoxDemo 819
TextBoxWithToolBar 953
TextColumns 382
TextOnBaseline 349
TextureBrushDemo 780
Thumbnail 458
TiltedShadow 904
ToggleButtons 958
Transform 876
TriangleGradientBrush 792
TriangleTile 796
TrimmingTheText 374
TryOneInchEllipse 231
TwentyFourPointPrinterFonts 335
TwentyFourPointScreenFonts 334
TwoButtons 556, 559
TwoButtonsAnchor 569, 577
TwoButtonsDock 572, 574
TwoForms 58
TwoPaintHandlers 72
TwoPanelsWithSplitter 1014
TwoPointLinearGradientBrush 785
TwoStatusBarPanels 934
TwoTriangleTile 801
TypeAway 220, 224, 313

U

UnderlinedText 370
Unicode 11, 190
UNIX 46

V

VGA (Video Graphics Array) 227
Visual Basic 19
Visual C# .NET 3, 39, 554
Visual Studio .NET 642

W

W3C (World Wide Web Consortium) 100
WarpText 920
WhatSize 244, 246
WhatSizeTransform 246
WidenPath 699
WidePolyline 683
WildCardHexDump 510
Win32 API 58, 139, 177, 339, 591, 686
Windows Forms 3
— клавиатура 201

— описатель окна 210
— цвет 99
— Designer 554, 590
Wink 471
WrapText 924

X
XMarksTheSpot 154, 159
XML (Extensible Markup Language) 4

А
альфа-канал 428
аргумент
— ClickEventArgs 952
— CombineMode 714
— DragEventArgs 1129
— DrawItemEventArgs 672
— DrawString 327
- EventArgs 293, 294, 385. 578. 582. 599, 603, 610, 747. 818, 867, 870, 876, 939, 1051
— fAngle 789
— FileMode 1076
- FillMode 637
— font. FontFamily 908
— font.FontStyle 908
— Format32bppArgb 713
- fSize 908
— GraphicsUnit 332
— ImageAttributes 780
- iOffset 636
— MatrixOrder 255
— MatrixOrder.Append 261
— MatrixOrder.Prepend 261
— MeasureItemEventArgs 672
- MetafileFrameUnit 1078
— MouseEventArgs 272
— PointF 910
— PrintEventArgs 977
— RectangleF 357, 701
— RotateTransform 252
— ScrollEventArgs 603
-Shear 901
— SplitterEventArgs 1021
— StringFormat 224, 361
— TreeViewCancelEventArgs 1028

Б
битовая карта 426
— двухуровневое (bilevel) изображение 427
— монохромное (monochrome) изображение 427
— полноцветная (full-color, true-color) 428
— формат файла 427
битовая маска 200
буксировка 1128
буфер обмена 1093
— программа просмотра 1114
— формат 1124

Д
делегат 42, 61, 155
— ClickEventHandler 952
— DragEventHandler 1129
— DrawItemEventHandler 672
- EventHandler 96, 190, 215, 293, 294. 385. 552. 578. 582, 599, 610, 654. 747, 818, 867, 870, 876, 939, 1051
— Graphics.EnumerateMetafileProc 1086
- KeyEventHandler 191
— KeyPressEventHandler 204
— MeasureItemEventHandler 672
— MouseEventHandler 272
— PaintEventHandler 62, 63
— PrintEventHandler 977
— SplitterEventHandler 1021
— TreeViewCancelEventHandler 1028
дерево 1023
— каталогов 1031
— событие 1027

И
индекс 27
индексатор 42
интернирование строк 18
интерфейс 42
— IButtonControl 721
— ICaptureLossNotify 290
— Icollection 662
— IEnumerable 19, 662
— IList 662
— графический 2
— клавиатурный 306
— командной строки 2
— пользователя 53
исключение 27
— ArgumentOutOfRangeException 30
— выброс 27, 29, 30
— генерация 27
— перехват 28

К
каноническая кривая 638
каскадная таблица стилей 100
квадратный пиксел 227
кисть 67, 100, 157, 766
- TextureBrush 886
— закрашивание текста 889
— мозаичное расположение 796
— с линейным градиентом 783
— с множественным градиентом 791
— текстурная 779
— узорная 767
клавиатура 188
— Microsoft Internet Keyboard Pro 198
— Microsoft Natural Keyboard Pro 198
— игнорируемая 188
— иноязычная 212
— интерфейс 202
— клавиши 190

— -ASCII 196
— — Microsoft 197
— — горячие 370
— — знаки действий 195
— — модификаторы 196. 200
— — нажатые 191
— — не имеющие символов 191
— — отпущенные 191
— — переключатели 190
— — просмотр 205
— — символьные 191
— — специальные 197
— — управления регистром 191
— — управляющие курсором 196
— — управляющие символы 204
— — функциональные 195
— — цифровые 194
— код символов 190, 198, 204
— комбинация клавиш 189
— раскладка 212
— фокус ввода 215
— цифровая 195
класс 5, 23, 31, 82
— AboutBox 730
— AboutDialogBox 730
— Application 56
— Array 147
— ArrayList 316, 317, 694
— ASCIIEncoding 493
— Bezier 631
— BezierClockControl 619, 621
— BezierManual 631
— Bitmap 426 460
— Brushes 67
— Button 551, 581
— ButtonBase 550, 559, 561
-Calendar 396
— CaptureLoss 288
— CaptureLossNotifyWindow 290
- CaptureLossWindow 288, 289
— Caret 219, 220
— CheckBox 550
— CheckerChild 308, 311, 412
- CheckerWithChildren 312
— CheckerWithKeyboard 306
— Clipboard 1096
— ClockControl 414
— Color 67
- ColorFillDialogBoxWithApply 736
— ColorScrollDialogBox 742
— CommonDlalog 744
— Console 7, 26, 45, 61
- ConsoleHelloWithUsing 7
— ConsoleHelloWorld 7
— ContainerControl 54, 311, 554
— ContextMenu 643, 662
— Control 53, 129, 155, 281, 575
— ControlPaint 151, 565
— Date 25, 26, 30
— DatePlus 38
— Debug 46
— DirectoryTreeView 1034
— Environment 51, 502
— Exception 28
— FileDialog 744
— FileStream 484
— Font 66, 118, 326
— FontFamily 326
— FontMenuForm 887, 896
— Form 53
— Graphics 65, 151
— GraphicsContainer 243
— GraphicsPath 685, 686, 688, 696, 916, 920
— GraphicsState 243
— Hashtable 316
— HatchBrush 101, 768
— HatchStyleMenuItem 773
— HelloWorld 78
— Image 426, 435, 457
— ImageFormat 438
— ImagePanel 1041
— InheritHelloWorld 79
— InheritWithConstructor 74
— InstantiateHelloWorld 80
— KeyPressEventArgs 204
— LinearGradientBrush 101, 783
— MainMenu 643, 662
— Marshal 16
— MarshalByRefObject 74
— Math 167, 522
— Matrix 259
— MemoryStream 1076
— Menu.MenuItemCollection 662
— MenuItem 643, 645, 654, 662, 939
— MenuItemColor 660, 664
— MessageBox 720
— MessageBoxHelloWorld 47
— Metafile 426
— NativeWindow 287
— Object 38, 40, 41
— PageSettings 973
— PaintEventArgs 63, 64, 68, 142, 143, 151
— Panel 133
— PathGradientBrush 101, 783
— Pen 101, 152
— PictureBox 480
— PieChart 180
— PrintableForm 887
— PrintController 988
— PrinterSettings 152, 966, 972
— PrintPageEventArgs 156
— QueryPageSettingsEventArgs 978
— Queue 316
— RadioButton 550, 598
— SampleClass 501
— SaveFileDialog 763
— ScrollableControl 53, 134. 135, 133, 139, 601
— SevenSegmentDisplay 409

— SimpleButton 552, 553
— SimpleControl 553
— SimpleDialog 719
— SolidBrush 101
— SortedList 316
— Splitter 1009
— Stack 316
- StatusBar 927
— StatusBarPanel 927, 935
— StreamReader 491
— StreamWriter 491
— StringBuilder 546
— StringCollection 968
— StringFormat 365
— SuperString 6
- SysInfoKeyboard 202
— SysInfoReflection 202
— SysInfoReflectionStrings 147
— SysInfoStrings 128, 129, 133, 142, 144
— SystemBrushes 104
— SystemColors 102
— SystemInformation 117, 144, 228, 269
— SystemPens 152
- TenCentimerRuler 234
-TextBoxBase 817
— TextureBrush 101, 783
— Thread 55
-Timer 7, 385
-ToolBar 927
— ToolBarButton 927
- TrackBar 609
— TreeNode 1027
- TreeNodeCollection 1023
-Type 41
— TypeAway 223
— UnicodeEncoding 493
— UserControl 308, 414
— UTF7Encoding 493
— UTF8Encoding 493
- Write 7
— абстрактный 101
— базовый 40
— встроенный 42
— изолированный 151
— имя 7
— параллельный 505
— файлового ввода-вывода 14
клиентская область 52, 69. 94, 108, 243
коллинеарная точка 622
командная строка 2, 46
компилятор 1, 8
компиляция 2
конкатенация 120
— строк 8
— числовой переменной 41
консоль 2, 44
константа 13
конструктор 34, 42, 132, 31б
— ArbitraryCoordinates 250
— BinaryReader 499
— BinaryWriter 499
— Bitmap 460, 470
— ContextMenu 644
— Cursor 470
— DataObject 1124
— DateTime 388, 394, 395
— Decimal 518
— DirectoryInfo 507
— FileInfo 512
— FileStream 485
— Font 326, 332, 344, 907
— FontFamily 345
— Graphics 151
— GraphicsPath 688
- HatchBrush 768
- HelloWorld 79
— Icon 468
— ImagePanel 1040
— LinearGradientBrush 784, 786, 789
— ListViewItem 1047
— ListViewItem.ListViewSubItem 1048
— MainMenu 644
— Margins 975
— Matrix 259
— MenuItem 644, 649
-Metafile 1060, 1071, 1078, 1080, 1084, 1085
— PageSettings 973
— PageSetupDialog 996
— PathGradientBrush 792
— Pen 152, 804
— PrintDialog 992
— PrintDocument 976
— PrinterSettings 967
— Rectangle 90
— RectangleF 90
— Region 714
— Size 87
— SolidBrush 767
— StreamReader 496
— StreamWriter 493
- String 532
— StringFormat 361
- TextureBrush 779
— TimeSpan 394
-TreeNode 1027
контур 179, 681, 686
— визуализация 692
— модификация 698
— отсечение 705
— преобразование 696
— создание 688
координатная точка 84
круговая диаграмма 179
курсор 216

M

мантисса 10
массив

-abChecked 305
— atile 423
— Controls 588
— float 12
— MenuItem 666
— MonthDays 27, 29
— PathPoints 687
— PathTypes 687
— Point 636
— PointF 636
— PropertyInfo 147
— string 588
— многомерный 13
— невыровненный 13
— точек 86
масштаб 256
матрица 258
межстрочный интервал 118
меню 52, 549, 642
— быстрого вызова 642
— быстрые клавиши 645
— всплывающее 643
— выпадающее 643
— главное 642
— дочернее 643
— контекстное 642, 658
— необычное 650
— подменю 643
— подсказка 939
— пункт 643
— — верхнего уровня 643
— — массив 643
— — набор 661
— — отмеченный 654
— создание 647
— стандартное 667
меню управления 56
метафайл 1058
— загрузка 1059
— запись 1085
— отображение 1059
— параметры отображения 1061
— преобразование 1068
— преобразование страниц 1080
— размер 1061
— создание 1071
— тип 1083
метод 5, 23, 26, 31, 42
-Add 664, 906, 946. 1023. 1045
— AddArc 685
— AddDirectories 1033
- AddFive 13
— AddLine 907
-AddPach 691
-AddRange 580, 935, 946, 1023, 1045
— AddString 906, 907, 908, 910, 924
— AddStrip 948
— Application.Run 57
— BeginContainer 243
— BezierSpline 632
— BringToFront 1010
— ButtonOnClick 552, 553, 558, 593, 598
— CanonicalSpline 641
— Caret 220
— Ceiling 89
— CheckBoxOnCheckedChanged 582, 586, 588, 601
— Clear 64, 100, 106, 1045
— Clipboard 1094
— Clipboard.SetDataObject 1125
— CloseFigure 686
— Color.FromArgb 100
— ComposeLine 1118
— Console.Read 46
— Console.ReadLine 46
— Console.Write 9
— Console.WriteLine 9, 28, 41
— Control 1133
— Convert 987
— Copy 16, 1088
— Cos 925
— CreateBitmapButton 965
— CreateChildren 312
— CreateGraphics 70, 129, 151
— CreateMeasurementGraphics 152
— CreateSampleFont 354
— CreateSubKey 752
— Cursor.Show 297
— DataFormats 1123
— DayOfYear 23, 25, 26, 29
— Directory.GetLogicalDrives 1033
— Dispose 129, 220
— Dolt 245
— DoPage 159, 180, 236, 247, 248, 342, 626. 693. 700, 706, 713, 887, 889, 895, 897
— Draw 906
— DrawArc 1073
— DrawBackground 678
— DrawBall 795
— DrawBezier 182
— DrawBeziers 182, 616, 621
— DrawClosedCurve 182, 636, 687
— DrawCurve 182, 632, 641
— DrawDots 415
— DrawEllipse 183, 1073
— DrawHourHand 619
— DrawImage 439, 1059, 1062
— DrawImageUnscaled 1059
— DrawItemEventArgs 674
— DrawLargerButton 568
— DrawLine 153, 1061
— DrawLines 165, 174, 632, 681, 810, 1084
— DrawMenuGlyph 674
— DrawPath 694, 907, 910
— DrawPolygon 173, 238, 687
— DrawRectangle 171, 172, 183, 238
— DrawRectangles 183
— DrawSmallerButton 568

— DrawString 65, 69, 114, 142, 158, 245, 337, 357, 361, 409, 897, 898. 904. 908. 1059
— DrawText 1 118
— EndContainer 243
— Enum.GetValues 560
- EnumerateMetafile 1085, 1086, 1087, 1088
- EnumMetafileProc 1086, 1088
— Equals 85, 86
— Exists 1074
-ExpandAll 1030
— FillClosedCurve 637
— FillElllpse 182, 183, 1073
— FillMode.Alternate 637
— FillMode.Winding 637
— FillPath 686, 694, 907, 914
-FillPie 183
— FillPolygon 184
— FillRectangle 182, 183, 223, 793
— FillRectangles 182, 183
— FillRegion 714
— Floor 523
— FontFamily 347
— Form 589, 721
— FromFile 1059, 1060
— FromImage 465, 1060, 1072
- Get 31
— GetAscent 889, 897, 902
— GetAutoScaleSize 589
— GetBounds 701, 910, 923
— GetData 1097, 1099, 1108, 1124
- GetDataObject 839, 1096, 1124, 1130
- GetDataPresent 1096, 1097, 1104, 1108, 1124
— GetDescent 889
— GetDirectories 1033
— GetFiles 1055
— GetFolderPath 51
— GetFontFamilyArray 355
— GetFormats 1107, 1118, 1124
— GetHdc 1071
— GetHeight 118, 119, 242. 245, 255, 276, 330, 337, 339
— GetItemAt 1051
— GetMetafileHeader 1062
— GetMonth 32
— GetNodeAt 1029
— GetProperties 147
— GetSysInfo 147
— GetThumbnailImage 458, 459
— GetType 41, 85, 1060
— GetValue 147, 753
— Graphics 255, 692
— Graphics.FromHwnd 152
— Graphics.FromImage 151, 453
— GraphicsPath 689, 690, 691
- HelloWorld.OnPaint 79
— Hide 220
— IButtonControl 724
-IDataObjecc 1096, 1108
- Image.FromFile 761, 1037, 1060
— InitializeCoordinates 415
— Insert 946, 1045
— Intersect 92
— Invalidate 96, 142, 284
- InvokeOnClick 598
— IsEmf 1063
- IsEmfOrEmfPlus 1063
— IsEmfPlus 1063
- IsEmfPlusOnly 1063
— IsLeapYear 25, 26, 34
- IsStyleAvailable 354
— KeyPress 204
— LinearGradientBrush 790
— ListBox 867
— ListView.ColumnHeaderCollection 1046
— ListView.ListViewItemCollection 1046
— ListViewItem.ListViewSubItemCollection 1047
— LoadRegistryInfo 854
— Main 25, 39, 76, 551, 719
— Math.Ceiling 133
— Math.Max 249
— Math.Min 143
— MatrixDialogOnChanged 879
- MaxLabelWidth 127, 1048
- MaxValueWidth 127. 133, 1048
— MaxWidth 127
- MeasureString 112, 127. 158, 223, 255, 313, 337, 360, 361, 406, 409, 456. 912. 1007
- MenuColorOnClick 735, 739, 749
— MenuFacenameOnDrawItem 678
— MenuFacenameOnMeasureItem 677
— MenuFilePreviewOnClick 1006
- MenuFilePrintOnClick 983, 984- 996, 999, 1006
— MenuFileSetupOnClick 999
— MenuOnClick 719
- MenuOnSelect 939, 942
— MetafileHeader 1063
— MetafileToBitmap 1068, 1070, 1071
— MMConv 238
— MoveTile 423
- MultiplyTransform 783, 905
— MyPaintHandler 64
-Offset 85
- On 215
— OnActivated 190
— OnAfterCollapse 1028
- OnAfterExpand 1028
- OnAfterSetect 1028
— OnApply 747
- OnBeforeCollapse 1028
— OnBeforeExpand 1028, 1034
- OnBeforeSelect 1028
— OnBeginPrint 977
— OnButtonClick 952
— OnButtonDropDown 952

— OnCheckedChanged 578, 599
— OnCheckStateChanged 582
— OnClick 155, 293, 553, 654
— OnClosed 755
— OnColumnClick 1051
— OnDeactivate 190
— OnDoubleClick 293
— OnDragDrop 1129
— OnDragEnter 1129
— OnDragLeave 1129
— OnDragOver 1129
— OnDrawItem 672
— OnEndPage 988
— OnEndPrint 977, 988
— OnGotFocus 216, 220
— OnHelpRequest 747
— OnItemActivate 1051, 1055
— OnKeyDown 191, 223, 553
— OnKeyPress 204, 223, 553, 598
— OnKeyUp 191, 553
— OnLostCapture 287, 288
— OnLostFocus 216, 220
— OnMeasureItem 672
— OnMenuComplete 939
— OnMenuStart 939
— OnMouseDown 272, 288, 314
— OnMouseEnter 294
— OnMouseHover 294
— OnMouseLeave 294
— OnMouseMove 272, 277, 284, 300
— OnMouseUp 272, 284, 644
— OnMouseWheel 272
— OnMove 96
— OnPaint 75, 106, 142, 143, 160, 580. 632, 781, 887, 1000
— OnPopup 654
— OnPrintPage 977, 999, 1007
— OnQueryPageSettings 977, 983
— OnResize 96, 309, 476, 557, 558, 1010, 1022
— OnScroll 603, 610
— OnSelect 654, 943
— OnSelectedIndexChanged 867, 870, 1051
— OnSelectedValueChanged 867
— OnSelectionChange Committed 870
— OnSplitterMoved 1021
— OnSplitterMoving 1021
— OnStartPage 988
— OnStartPrint 988
— OnTextChanged 818, 870
— OnTick 385
— OnValueChanged 603, 610, 876
— OpenSubKey 752
— PanelOnPaint 133, 960, 961, 965, 1118
— Path.GetFileNameWithoutExtension 763
— PlayRecord 1088
—- PointsToPageUnits 908
—- PointToClient 98, 297. 1130
—- PointToPageUnit 889
—- PointToScreen 98
— PositionCaret 223, 224
— PreviewPrintController 1001
— Print 156, 977
— PrintableForm 234
— PrintController 988
— PrintDocument 977
— PrintDocumentOnPrintPage 155, 156. 158, 159, 160
— PrinterSettings 973
— PrintPage 983, 1007
— Process.Start 1055
— RadioButtonOnCheckedChanged 601
— Randomize 419
— Read 61, 1110
— ReadByce 1110
— ReadLine 61
— RefreshTree 1033. 1036
— RegistryKey 752
— ReleaseHdc 1071
— Remove 946, 1045
— ResetTransform 255, 260, 783
— Restore 242, 245
— RotateTransform 255, 260, 783, 898
— Round 89
— RoundedRectangle 178
— Run 56
— Save 242, 245, 457
— SaveRegistryInfo 854
— ScaleCore 591
— ScaleImageIsotropically 444, 451. 1000
— ScaleTransform 254, 255, 260, 783, 898
— ScrollLines 211
— SendToBack 1010
— Sec 31
— SetChildIndex 1010
— SetClip 705, 709, 714
— SetData 1124, 1125
— SetDataObject 1099, 1104, 1124
— SetMonth 30. 32
— SetValue 753
— Shear 900, 901
— Show 48, 54, 220, 720
— ShowDialog 720, 736, 993, 999, 1001
— ShowFiles 1055, 1057
— ShowImages 1040, 1044
— Sin 925
— Size.Ceiling 133
— Sleep 991
— Sort 147
— StandardPrintController 990
— String.Format 9, 41
— StrokePath 686
— Substring 677
— SysInfoStrings 142
— TextBoxBase 821, 830, 838
— ToKnownColor 608
— ToolBarOnClick 956, 961
— ToolTip 856
—• ToSize 89
—• ToString 40, 41, 85, 607

— Transform 696
— TransformPoints 245, 246, 247
- TranslateClip 711, 713
— TranslateTransform 254, 255, 260, 568, 696, 783, 898, 903, 905, 910, 911
— TreeNode 1030
- TreeNodeCollection 1023
— TreeView 1026, 1030
— Truncate 89
- UpdateAllInfo 142
— Warp 701, 920, 921
— Widen 698, 916
— WriteLine 7, 26
— переопределение 41
— по значению 13
— статический 26
метрические единицы 243
многострочный комментарий 4
модель ввода 61
модификатор
— new 41
— override 41
— private 30
— static 26
— virtual 41
— доступа 25, 41
мышь 267. 268
— горячая точка 268
— движение 277
— захват 279, 281
— игнорируемая 268
— колесико 270
— курсор 268, 296, 303
— определение позиции курсора 304
— отслеживание 279
— свойства 294
— событие 271

О

область 714
— вырезанная 161
обработчик события 61
— ButtonClick 961
- CheckedChanged 579, 580
— Click 550, 552
— ColorFillDialogOnApply 739
— DrawItem 673
— EventHandler 603
— GotFocus 220
— KeyDown 553
— KeyPress 204
— KeyPress 553
— KeyUp 553
— LostFocus 220
— MeasureItem 672
— MenuFacenameOnPopup 666
— OnResize 569
— PrintDocumentOnPrintPage 156
— PrintDocumentOnPrintPage 156
— PrintPage 156
— ScrollEventHandler 603
— Tick 1041
-TreeView 1029
общее линейное преобразование плоскости 257
объект 23, 26
— ArrayList 316
— Bitmap 1105
— Brush 101
— btnLarger 558
— btnSmaller 558
— Button 550
— Capture Loss Window 288
— CheckBox 733
— Color 106
— ColorScrollDialogBox 743
— ContextMenu 659, 663
- Date 25, 29
— DatePlus 41
- DateTime 394
— DrawItemEventArgs 151, 673
— FileStream 486
— Font 127
— FontFamily 344
— FontStyleCheckBox 588
-Graphics 68, 127, 142, 230, 242, 552
— Graphics 889
- GraphicsPath 686, 706
— GroupBox 733
— ImageList 946, 948, 965
— InputLanguageChangingEventArgs 215
— KeyEventArgs 193, 199
— MainMenu 644, 667
— Matrix 161
— MeasureItemEventArgs 151
— MemoryStream 1076
— MenuItem 662, 677
— MenuItemColor 660
— MessageBox 48
— Metafile 1105
— mydate 26
— PageSettings 975, 985
— Panel 931
- PathGradientBrush 795, 802
-Pen 152, 804
— PictureBoxPlus 1041
— PreviewPrintController 988
— PrintControllerWithStatusDialog 1000
- PrintDocument 155, 156, 978, 985, 988
— PrinterSettings 967, 985
- PrintPageEventArgs 151, 156, 157, 985
— PropertyInfo 147
— RadioButton 598
-Rectangle 98, 317, 984
— ScrollableControl 203
— SimpleButton 551
— StatusBar 929, 935
- StatusBarPanel 933, 935, 942
— String 1105
— StringFormat 361, 364

— ToolBarButton 956
— ToolTip 856
— VscrollBar 60S
однострочный комментарий 4
оператор 14, 42
— break 17, 19
— Console.Write 46
— Console.WriteLine 46
— continue 19
— delete 13
- do 17
— for 18
— foreach 19, 351
— goto i8, 19
- if 15
— new 1 2
— return 32
— switch 18
— typeof 41
— using 6, 54
- while 17
— аддитивный 14
— вычитания 40
— группировка I
— логический 15
— логическое И 14
— логическое ИЛИ 14
— логическое исключающее ИЛИ 14
— мультипликативный 14
— отношение 14
— перегрузка 40
— побитовый 14
— присваивание 8, 15
— простой 14
— равенство 14
— сдвиг 14
— сравнения 40
— унарный 14
— условие 1 5
— условное И 14
— условное ИЛИ I 5
оптимизированная палитра 427
отражение 144
отсечение 681
ошибка смещения 183

П

панель задач 94
панель инструментов 927
— простая 945
— событие 952
— стиль 957
параметр
— /main 39
— amiFile 645
— ArgumentOutOfRangeException 3 1
— object 552
— PageUnit 445
— PrintPageEventArgs 1 56
— Reference files 144
- WrapMode.Clamp 786
— year 26
параметрическое уравнение 168
параметры экрана 229
переключатель 549
переменная
- arr I 2
— bBlocking 283
— bValidBox 283
— cxBtn 573
— cxCol 129. 133
— cyBtn 573
— cyText 580
— dxBtn 573
— form 83
— fSlice 181
— iFirst 143
- iLast 143
— mydate 23, 26
— str 85
— string 12
— trigger 16
— видимости 297
перечисление 42
- AnchorStyles 569
— Appearance 581
- BorderStyle 480, 584. 585
— ButtonState 565
— CharacterCasing 855
— CheckState 582
— ColorDepth 477
— CombineMode 707, 708
— ComboBoxStyle 869
— ContentAlignment 559
— CoordinateSpace 245
— DashCap 812
— DashStyle 807
— DialogResult 50, 720
— DockStyle 569, 572, 1009
— DragAction 1133
— DragDropEffects ИЗО
— DrawItemState 673
- DrawMode 869
— Duplex 972
- - EmfPlusRecordType 1083
— EmfType 1084 '
— Environment.SpecialFolder 51, 503
— FileAccess 14, 485
— FileAttributes 509
— FileMode 14, 485
- FileShare 485, 486
— FillMode 185, 637, 687
— FlatStyle 560
— FontStyle 326, 585, 691
— FormBorderStyle 60
— FormStartPosition 60, 725
— FormWindowState 826
— GenericFontFamilies 346
— GraphicsUnit 234, 333
— HatchStyle 769, 774

— HorizontalAlignment 823, 936, 1046
— HotkeyPrefix 370
— ItemActivation 105 1
— Keys 192, 193
— KnownColor 104, 608
— LinearGradientMode 787
— LineCap 814
— LineJoin 810
— MatrixOrder 255, 696
— MenuGlyph 674
— MessageBoxButtons 49
— MessageBoxDefaultButton 50
— MessageBoxIcon 49
— MessageBoxOptions 50
— MetafileFrameUnit 1078
— MetafileType 1062
— MouseButtons 272
— Orientarion 609
— PaperKind 970
— PaperSourceKind 970
— PathPointType 688
— PenAlignment 806
— PictureBoxSizeMode 480
— PixelFormat 437
- PixelOffsetMode 163
— PrinterResolutionKind 971
— PrinterUnit 987
— PrintRange 993
— RichTextBoxScrollBars 823
— RotateFlipType 459
— ScrollBars 822
— ScrollEventType 603
— SeekOrigin 487
— SelectionMode 865
— Shortcut 645
— SmoothingMode 163
— SpecialFolder 5 I
— StatusBarPanelAutoSize 937
— StatusBarPanelBorderStyle 936
— StatusBarPanelStyle 935
- StringAlignment 109, 366
— StringFormatFlags 362
— StringFormatFlags.MeasureTrailingSpaces 224
— StringTrimming 374
— TextRenderingHint 359
— TickStylc 610
— ToolBarAppearance 950
- ToolBarButtonStyle 957
- ToolBarTextAlign 95 1
— TreeViewAction 1028
— View 1045
— Warp Mode 702
— WrapMode 779
перо 100, 152, 157, 766, 804
— пунктир 807
— толщина 238
печать 154, 966
— диалоговое окно 992
— область I 57
— определение документа 976
— предварительный просмотр 1000
— принтер 157
- размер бумаги по умолчанию 156
— размер страницы 984
подкласс 38
поле 26, 31, 42
— Empty 85
— MaxValue 43
— MinValue 43
— month 3 1
— ввода с форматированием 85 5
— редактирования 817
— текстовое 817
— — многострочное 822
— — однострочное 817
— подстановки 8
— редактирования 53
ползунок 130, 602
полилиния 321
полоса прокрутки 3, 53, 130, 549, 601
последовательный опрос 61
преобразование
— глобальное 250
— единичное 257
— комбинирование 264
— линейное 256
— матричное 259
— сдвиг 26 I
префикс 43, 96
— Draw 182
— Fill 182
приведение типов 16. 42. 501
приглашение MS-DOS 2
принтер 231, 966
приращение 256
проверка допустимости значений 34
проект 3
пространство имен 5. 6
-System 7, 28, 41, 84, 388
— System, System.Drawing 7
- System. Environment 502
— System.Collections 19, 316
— System.Diagnostics 46
— System.Drawing 59. 63. 66, 82, 426. 559, 714
— System.Drawing.Drawing2D 163. 243. 255, 259, 806
— System.Drawing.Imaging 436, 1083
— System.Drawing.Printing 1 55, 967. 972
— System.Globalization 396
— System.IO 484
— System.Runtime.InteropServices 10
— System.Threading 55
— System.Timers, System.Threading 7
— System.Windows.Forms 7, 48, 53, 56, 62. 117. 296, 477, 564, 643. 966, 1001
— глобальное 7
прямоугольник 90, 171
псевдоним 7

Р

рабочий стол 94
равенство 86
развертывание 825
раздел 751
разделитель 1008
разрешение дисплея 228
растр 424
расширенный набор символов 1 2
регистр 5, 39
регулятор 609
реестр 751
решение 3
родитель 54

С

сборка мусора 13
свертывание 825
свойство 31. 42
— AcceptButton 724, 728
— AcceptsReturn 823
— AcceptsTab 823
— Action 1028, 1133
— Activation 1050
— Active 857
— ActiveBorder 102, 103
— ActiveCaption 102, 103
— ActiveCaptionText 102, 103
— ActiveForm 189
— AddExtension 763
— AliceBlue 101
- Alignment 109. 365, 367, 805, 806, 936
— AllowDrop 1129
- AllowedEffect 1130
— AllowFullOpen 748
— AllowMargins 996
— AllowOrientation 996
— AllowPaper 996
— AllowPrinter 996
— AllowPrintToFile 992
— AllowSelection 992
— AllowSome Pages 992
— Anchor 569, 820
— AntiqueWhite 101
— AnyColor 748
— Appearance 581, 949
— AppStarting 296
— AppWorkspace 102, 104
— ArrangeStartingPosition 121
— Arrow 296
- ASCII 493
— Attributes 508, 512
- AutoCheck 578, 598
— AutoFlush 495
— AutomaticDelay 857
— AutoPopDelay 857
— AutoScale 589, 591
— AutoScaleBaseSize 589, 590, 593
— AutoScroll 130, 134, 138, 930, 933
— AutoScrollMargin 134
— AutoScrollMinSize 135, 138
— AutoScrollPosition 135, 138, 203
— AutoSize 584, 609, «20, 936
— BackColor 59,67, 100, 105, 558, 610, 673, 929, 1020
— BackgroundColor 769
— BackgroundImage 610
— BarBreak 653
— BaseStream 495
— BigEndianUnicode 493
— • Blend 791
— Bmp 429
— Bold 339
— BorderStyle 584, 936, 949, 1020
— Bottom 91, 93, 975
— Bounds 93, 673, 974. 1063
— Break 653. 654
— Brush 806
— Button 952
— Buttons 946
— Cancel 978, 1028
— CancelButton 724, 728
— CanDuplex 971
— CanFocus 215
— CanRead 486
— CanSeek 486
— CanUndo 838
— CanWrite 486
— Capture 281
— CenterColor 793
— CenterPoint 793
— CharacterCasing 854
— Check 666
— CheckAlign 581
— Checked 578. 598, 655
— CheckedChanged 578
— CheckFileExists 7 59
— CheckPathExists 759
— CheckState 582
— Chocolate 77
— ClassesRoot 752
— ClientRectangle 173
— ClientSize 94, 138, 153, 156, 243
— Clip 161, 296, 714
— ClipBounds 714
— ClipRectangle 63, 142. 143
— Collate 971
— Color 101, 746, 747, 767, 806, 974
— ColorDepth 477
— Column 1051
— Columns 1045
— ContainsFocus 215
— ContextMenu 644, 664
— Control 102, 103, 104, 219
— ControlDark 102, 103, 104
— ControlDarkDark 102, 103, 104
— ControlLight 102, 103, 104
— ControlLightLight 102. 103, 104
— Controls 575, 580
— ControlText 102, 103, 104

— Copies 971
— Count 663, 968, 1023, 1046
— CreatePrompt 763
— CreationTime 508, 512
— Cross 296
— Current 296
— CurrentConfig 752
— CurrentDirectory 504
- CurrentInfo 399
— CurrentUser 752
— Cursor 60
— Cursor.Current 300
— Cursor.Position 618
— CustomColors 749
— CustomEndCap 816
— CustomStartCap 816
— DashCap 810
— DashOffset 807, 809
— DashPattern 807, 809
- DashStyle 807
-Date 389
-Day 32, 389
— DayOfWeek 389
- DayOfYear 34, 389
- Days 394
— DaysSince1600 38
— DecimalPlaces 876
- Default 296, 493
— DefaultItem 653
— DefaultPageSettings 972, 976
— Desktop 102, 104
— DesktopBounds 94, 825
— DesktopLocation 94
- DiatogResult 720, 721, 724
— Directory 512
— DirectoryName 512
— Divider 949
-Dock 569, 572, 931, 933, 1013
— Document 992, 996, 1001
— DocumentName 155, 976
— DoubleClickSize 269
— DoubleClickTime 269
-DpiX 230, 241, 335, 1063
-DpiY 230, 241, 335, 1063
— DrawItemEventArgs 674
— DropDownArrows **962**
— DropDownMenu 957
— DropDownStyle 869
— DroppedDown 869
— Duplex 971
— DynData 752
-Effect 1130
— Elements 260
-Emf 429
— EmfPlusHeaderSize 1062
- Enabled 385, 554, 653, 957
— Encoding 495
-EndCap 810
— Entries 438
— EscapePressed 1133
— Exif 429
-Exists 507, 512
— Extension 507, 512
— Families 356
— FileName 756
— FileNames 756
— FillMode 687
- Filter 761
— FilterIndex 761
— FirstNode 1029
— FixedPitchOnly 747
-Flags 438
- FlatStyle 560
-Focused 215, 312
-Font 70, 119, 158, 557, 558, 580, 673, 746
- Font.Height 588
— FontFamily 339
— FontHeight 119, 325
-ForeColor 100, 105, 152, 160, 558. 673, 929
— ForegroundColor 769
— FormatFlag 224
-FromPage 972, 993
-FullName 507, 512
- FullOpen 748
— FullPath 1030
— GammaCorrection 791
- GdiCharSet 339
— GdiVerticalFont 339
— GenericDefault 362
— GenericMonospace 346
— GenericSansSerif 346
— GenericSerif 346
— GenericTypographic 362
— Gif 429
-Graphics 63, 672, 673, 978
— GrayText 102, 103
— Guid 438
-Hand 296
-Handled 204
— HasMorePages 156, 978
— HatchStyle 769
-Height 60, 87, 91, 93, 113, 119, 156, 339, 557, 573, 970, 1061
-Help 296
— HelpButton 1001
— Hexadecimal 876
- HideSelection 820
— Highlight 102, 103, 104
- HighlightText 102, 103, 104
— HorizontalResolution 436, 1061, 1081
— HorizontalScrollbar 865
- HotkeyPrefix 370
— HotTrack 102, 104
— Hour 389
-Hours 394
-HScroll 134
- HSplit 296
— IBeam 296

— Icon 429, 935
—-Id 1123
— Image 561. 584, 782, 1001
—- ImageAlign 561, 584
— ImageIndex 561, 584. 946, 1027
— ImageList 561, 584, 946, 1027
—- Images 477
—- ImageSize 477, 946
—- InactiveBorder 102, 104
—- InactiveCaption 103, 104
—- InactiveCaptionText 103
—-Increment 875
—- Index 662, 672, 673, 1029
—-Info 103. 104
— InfoText 103
—- InitialDelay 857
—- InitialDirectory 758
— InstalledPrinters 968
—- IntegralHeight 865
—- InterpolationColors 791
—-Interval 385
—- InvariantInfo 399
— IsClipEmpty 714
— IsDefaultPrinter 967
— IsEmpty 87. 91
— IsExpandcd 1030
—- IsIdentity 260
—- IsInvertible 260
— IsParent 662
—- IsPlotter 969
— IsReadOnly 1023, 1045, 1046
—- IsSelected I 030
—- IsValid 967
— IsVisibleClipEmpty 715
— Italic 339
— ItemHeight 672
— Items 864, 870. 1045
— ItemWidth 672
— Jpeg 429
— KeyChar 204
— KeyCode 192. 199, 200
— KeyData 200
— KeyState 1129, 1133
— Kind 970
—-Labels 127
—- Landscape 974
—- LandscapeAngle 969
—- LargeChange 602, 609
—- LargeImageList 1045
—- LastAccessTime 508, 512
— LastNode 1029
— LastWriteTime 508, 512
— Left 91, 93, 975
— Length 12, 43, 486, 512
—- LineAlignment 109, 365
—- LinearColors 789
—- LinearGradientMode 805
— LineJoin 810
— Lines 818
—- Local-Machine 752
— Location 93, 98, 133. 580, 589, 728
— LogicalDpiX 1003
— LogicalDpiY 1063
— MarginBounds 157, 979
— Margins 974
— MaximizeBox 60
— Maximum 134, 602, 607, 609, 875
— MaximumCopies 971
— MaximumPage 972, 993
— MaxLength 818
— MaxWindowTrackSize 139
— MemoryBmp 4 29
— Menu 103, 104, 644
— MenuItems 662
— MenuText 103
— MetafileSize 1062
— Millisecond 389
— Milliseconds 394
— MinExtra 1020
— Minimum 134, 602, 609, 875
— MinimumPage 972, 993
— MinMargins 996
— MinSize 1020
— Minute 389. 415
— Minutes 394
— MinWidth 936, 937
— MiterLimit 812
— Mnemonic 653
— Modified 829
— ModifierKeys 201, 294
— Modifiers 200. 202
— Month 31, .32, 389
— MouseButtons 269. 294
— MouseButtonsSwapped 269
— MousePosition 294, 296
— MousePresent 269
— MouseWheelPresent 270
— MouseWheelScrollLines 270
— Multiline 822
— Multiselect 756
— MultiSimple 866
— Name 147, 339, 344, 507, 5 | 2. 1123
— NativeMouseWheelSupport 270
— NewLine 495
— NewValue 603
— NextNode 1029
— NextVisibleNode 1029
— No 296
— Node 1028
— Nodes 1023, 1024
— NoMove2D 296
— NoMoveHoriz 296
— NoMoveVert 296
— Now 389
— OffsetX 260
— OffsetY 260
— Orientation 609
— OverwritePrompt 763
— Owner 742
— OwnerDraw 151, 672

— PageBounds 156, 979
• PageScale 161, 234, 236, 241, 249, 250, 255
- PageSettings 978, 996
- PageUnit 161, 234, 236, 241, 248, 250, 255
-Palette 438
- PanEast 296
— Panels 933, 935
- PanNE 296
— PanNorth 296
- PanNW 296
— PanSE 296
- PanSouth 296
- PanSW 296
— PanWest 296
— PaperName 970
-PaperSize 974, 975
— PaperSizes 969
— PaperSource 974, 975
— PaperSources 969
- Parent 507, 662, 927, 935, 1029
— PartialPush 957
— PasswordChar 854
— PathData 687
— PathPoints 687
— PathSeparator 1030
— PathTypes 687
- PenType 806
— PerformanceData 752
— PhysicalDimension 436, 1061
- PhysicalSize 1001
— PixelFormat 436, 453
- PixelOffsetMode 161, 163, 358
-Png 429
- Point 91
— PointCount 687
- Points 687
— Position 219, 223, 296, 486
— PreferredHeight 584, 865
— PreferredWidth 584
— PrevNode 1029
— PrevVisibleNode 1029
— PrimaryMonitorMaximizedWindowSize 139
— PrimaryMonitorSize 139
— PrintController 976, 988, 1001
— PrinterName 967
— PrinterResolution 974, 975
— PrinterResolutions 969
- PrinterSettings 966, 974, 976, 992, 996
— PrintPreviewControl 1001
— PrintRange 972, 993
— PrintToFile 972, 992, 993
— Pushed 957
— RadioCheck 655, 659
— RawFormat 438
- ReadOnly 854
— ReadOnlyChecked 759
— Rectangle 789, 796
— RenderingOrigin 777
— ReshowDelay 857
— ResizeRedraw 107
— RestoreDirectory 758
-Right 91, 93, 975
— Root 507
-Rtf 856
- ScrollAlwaysVisible 865
- ScrollBar 103, 104
— ScrollBars 822
— Second 389, 415
— Seconds 394
-Selected 1052
— SelectedImageIndex 1027
- SelectedIndex 865, 866, 870
— SelectedIndices 865, 1051
- SelectedItem 865, 866, 870
— SelectedItems 865, 1051
— SelectedNode 1030
— SelectedRtf 856
- SelectedText 820
— SelectionLength 820
— SelectionMode 865
— SelectionStart 820
— Shortcut 653
— ShowAlways 857
— ShowApply 746
— ShowColor 746
— ShowEffects 746
— ShowHelp 746, 748, 992, 996
— ShowLines 1026
- ShowNetwork 992, 996
— ShowPanels 933
— ShowPlusMinus 1026
— ShowReadOnly 759
— ShowRootLines 1026
— ShowShortcut 653
— ShowToolTips 946
— Size 91, 93, 98, 219, 339, 580, 589, 1061
- SizeAll 296
— SizeInPoints 339
— SizeMode 730
— SizeNESW 296
- SizeNS 296
— SizeNWSE 296
- SizeWE 296
— SizingGrip 929
— SmallChange 602, 609
— SmallImageList 1045
— SmoothingMode 161, 163, 358
- SolidColorOnly 748
— Sorted 864, 1026
— SourceName 970
— SplitPosition 1020
-SplitX 1021
— SplitY 1021
— StartCap 810
— StartPosition 60, 725
— State 673
— Strikeout 339

— StringFormat 383
— Style 339, 935, 957, 961
— SupportsColor 969
— SurroundColors 793
— SystemDirectory 504
— SystemInformation 133
— SystemInformation.CaptionHeight 138
— SystemInformation.DragFullWindows 139
— SystemInformation.FrameBorderSize 138
— SystemInformation.MinimumWindowSize 250
— SystemInformation.VerticalScrollBarWidth 138
— TabIndex 585
— TabStop 585, 608, 609, 930
— Tag 586, 953, 1029
— Text 56, 653, 818, 935, 942, 951, 1029, 1046
— TextAIign 559, 581, 823, 949, 1046
— TextLength 818
— TextRenderingHint 358
— ThousandsSeparator 876
— ThreeScate 582
— TickFrequency 610
— Ticks 393, 394
— TickStyle 610
— Tiff 429
— TimeOfDay 393
— Today 389
— ToolTipText 935, 936, 946
— Top 91, 93, 975
— ToPage 972, 993
— Top Index 864
— TotalDays 394
— TotalHours 394
— TotalMilliseconds 394
— TotalMinutes 394
— TotalSeconds 394
— Transform 161, 260, 782, 783, 789, 805
— Transparent 101
— TransparentColor 477
— TreeView 1029
— Trimming 373
—•Type 603, 1062
— Types 687
— Underline 339
—• Unicode 493
—-Unit 339
— UpArrow 296
—• UseAntiAlias 1000, 1001
— UseMnemonic 584, 585
—• Users 752
— UtcNow 389
— UTF7 493
— UTF8 493
—-Value 602, 603, 609, 875
—• Values 127
—-Version 1062
- VerticalResolution 436. 1061, 1081
— View 1045
— VirtualScreen 139
— Visibility 219
— Visible 57. 554, 653, 957
— VisibleClipBounds 157, 158, 243. 245. 248, 715, 984
- VScroll 1 34
— VSplit 296
— WaitCursor 296
-Width 60, 87, 91, 93, 113. 152. 156. 805, 935, 936, 970, 1020, 1046, 1061
— Window 103, 104
— WindowFrame 103
— WindowState 826
- WindowText 103. 104
— Wmf 429
- WmfHeader 1062
— Wordwrap 822
— WorkingArea 139
- WrapMode 782, 786, 789, 796
— Wrappable 949
- Year 32, 389
-Yellow 101
— YellowGreen 101
— неизменяемое 32
сглаживание 161
сегмент кривой 632
сектор 175
сигнатура 289
система координат диалогового окна 591
системный цвет 102
скобки
— простые 1
— угловые 1
— фигурные 1, 5
событие 42, 61
— Activated 190
— AfterCollapse 1028
— AfterExpand 1028
— AfterSelect 1028
-Alt 192
-Apply 747
— BcforeCollapse 1028
- BeforeExpand 1028
— BeforeSelect 1028
— BeginPrint 977
— ButtonClick 952
— ButtonDropDown 952
— ButtonOnClick 552
- Changed 879
- CheckedChanged 578. 599
— CheckStateChanged 582
- Click 293, 552, 558, 564, 654
— ColumnClick 1051
— Control 192
— Deactivate 190
— DirectoryTreeViewOnAfterSelect 1057
— DisplaySettingsChanged 140, 466

- Doubleclick 293
-DragDrop 1129
-DragEnter 1129
— DragLeave 1129
— DragOver 1129
- DrawItem 151, 672, 936
— EndPrint 977
— GotFocus 216
-Handled 192
— HelpRequest 747
— InputLanguageChanged 215
— InputLanguageChanging 215
— ItemActivate 1051
— KeyCode 192
— KeyData 192
- KeyDown 191, 200, 554
— KeyPress 204, 554
— KeyUp 191, 200
— KeyValue 192
— LostFocus 216
— MeasureItem 151, 672
— MenuComplete 939
— MenuFileOpenOnClick 645
— MenuStart 939
-Modifiers 192
— MouseDown 272, 293
— MouseEnter 294
— MouseHover 294
— MouseLeave 294
- MouseMove 272, 277, 293
— MouseUp 272, 293
— MouseWheel 272
— Move 96
-Paint 62, 63, 257
— Popup 654, 666
— PrintPage 155, 257, 977
— QueryPageSettings 977
— Resize 96
-Scroll 603, 604, 610
-Select 654, 942
— SelectedIndexChanged 867, 870, 1051
— SelectedValueChanged 867
— SelectionChangeCommitted 870
— SelectionChanged 957
— Shift 192
— SplitterMoved 1021
— SplitterMoving 1021
— TextChanged 818, 870
- Tick 385
— UserPreferenceChanged 140
- ValueChanged 603, 604, 608, 610, 876
сообщение об ошибке 27
список 549, 1045
сплайн 614
— канонический 632
— натяжение 632
среда программирования графики
с неопределенным состоянием 161
среднее качество цветопередачи 428
стек 27
стиль
— верблюда 43
— языка Pascal 43
страница
— единицы измерений 234
— масштабирование 234
— преобразование 241, 242
строка состояния 927
— и автоматическая прокрутка 930
— панель 933
— простая 928
строка форматирования 8
строковая константа 18
строковая переменная 18
структура 42, 82
- Color 99
— Date 22
- Point 87, 153, 166, 203, 247
— PointF 88, 114, 153, 166, 913
— Rectangle 90, 156, 678
-RectangleF 90, 157, 172, 701, 913
— Size 86
— SizeF 89, 112
структурная обработка исключений 27
стыковка 572
субконтур 686
счетчик с прокруткой 875

T

табуляция 378
текст
— ввод справа налево 224
— выравнивание 365
— высота 361
— горизонтальная ориентация 109
— закрашивание 889
— запись 491
— независимость от устройства 226
— отображение 220
— отсечение 371
— подгонка 371
— подгонка под координатную
сетку 363
— позиционирование на странице 2
— позиция курсора 313
— показ 65
— поле ввода 3, 549
— преобразование 363
— размер строки 360
— редактирование 220
— сглаживание 358
— точка вставки 224
— форматированный 1
— центрирование 112
— чтение 491
— ширина 361
текстовая строка 5
текущая позиция 160
телетайп 2

У
узел 1023
управляющие стрелки 875
управляющий элемент 3
условие 16

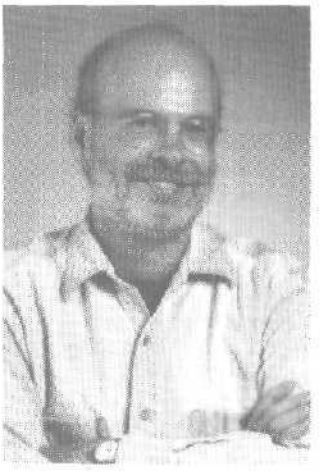
Ф
фигура 686
— закрытая 686
— открытая 686
флажок 549, 578
фокус ввода 189
форма 52, 215
— AutoScrollPosition 143
— SimpleButton 553
— дочерний элемент 575
— клиентская область 94
— конструктор 138
— наследование 73
— отображение 54
— размер 93, 107
— свойства 59
— создание 56
форматирование
— Currency (денежные суммы) 9
— Decimal (десятичный) 9
— Exponential (экспоненциальный) 9
— Fixed point (с фиксированной точкой) 9
— General (общий) 9
— Hexadecimal (шестнадцатеричный) 9
— Number (числовой) 9
— Percent (процент) 9
формула Безье 628
фрейм 1009
функция
— DayOfMonth 22
— DayOfYear 21, 23
— DrawLine 161
— DrawString 161, 252
— GetFolderPath 503
— IsLeapYear 22
— main 1. 22
— MoveTo 161
-printf 1, 2, 9
— RoundRect 177
— ScrollDC 139
- ScrollWindow 139, 210
— ScrollWindowEx 139
— sprintf 9
— System.Console.WriteLine 5
— WriteLine 8

Ц
цвет, выбираемый пользователем 102
целочисленное переполнение 516
цикл 16
— do 19
— for 18, 19, 186
— foreach 19, 147
— while 19

Ш
шаблон диалогового окна 591
шрифт 66, 320
— векторный 320
— высота 324
— гарнитура 321
— кегль 321
— коллекция 356
— контурный 150, 321, 887
— массивы гарнитур 350
— межстрочный интервал 324
— метрика дизайна 346
— название 321
— наклон знаков 322
— начертание 326
— плоттерный 320
— по умолчанию 325
— преобразование 896
-размер 229, 332
— разметка 321, 908
— растровый 320
— создание 328
— ширина знаков 322
— штрих 322
— штриховый 320

Э
экземпляр 23, 26, 38
экранные координаты 98
элемент управления 53, 215, 550
— CheckBox 578, 581, 586
— ComboBox 863
- GroupBox 598
- Label 308, 730
— ListView 1045
— Panel 308
— PictureBox 730
— PrimaryMonitorSize 228
— RichTextBox 216, 817
— TextBox 216, 220, 224
-ToolBar 945
— TreeView 1023
— z-порядок 577
— автоматическое масштабирование 589
— идентификация 585
— контейнер 554
— целевой 1009
—«дерево» 1008
—«надпись» 582
—«список» 1008
эллипс 174

Чарльз Петцольд



Чарльз Петцольд (*www.charlespetzold.com*) с 1985 г. разрабатывает программы для Microsoft Windows. Примерно с того же времени пишет на эту тему статьи и книги. Его первая статья о Windows-программировании была напечатана в журнале *MicrosoftSystemsJournal* в декабре 1986 г. Его книга «Programming Windows» (опубликованная Microsoft Press в 1988 г.) выдержала пять изданий и научила целое поколение программистов писать программы для Windows. В мае 1994 г. Петцольд стал одним из семи лауреатов (при этом он был единственным писателем среди них) премии "Windows Pioneer Award" журнала *Windows Magazine* и корпорации Microsoft за вклад в развитие Microsoft Windows. Чарльз также является автором уникальной работы по основам вычислительной техники — книги «Код» (М: Русская Редакция, 2001). Сейчас Петцольд работает над книгой по истории программирования.

Петцольд Чарльз

Программирование для Microsoft Windows на C#

Том 2

Перевод с английского под общей редакцией *В. Г. Вшивцева*

Переводчики: *С. А. Аристов, В. В. Краснов, Ю. С. Левашко, К. А. Меньшов, Н. Б. Соловченко*

Технический редактор *С. В. Дергачев*

Компьютерная верстка *В. Б. Хильченко*

Дизайнер обложки *Е. В. Козлова*

Фотография Ч. Петцольда на обложке *Deidre Sinott*
(фото опубликовано на сайте *www.charlespetzold.com*)

Оригинал-макет выполнен с использованием
издательской системы Adobe PageMaker 6.0

TypeMarket FontLibrary
легальный пользователь

ПОЛЬЗОВАТЕЛЬ
Para(-)Type
LEGAL USE

Главный редактор *А. И. Козлов*

Подготовлено к печати издательством «Русская Редакция»
121087, Москва, ул. Заречная, д.9
тел.: (095) 142-0571, тел./факс: (095) 145-4519
e-mail: info@rusedit.ru, http://www.rusedit.ru

РУССКАЯ РЕДАКЦИЯ

Подписано в печать 25.07.02 г. Тираж 3 000 экз.
Формат 70x100/16. Физ. п. л. 39

Отпечатано в ОАО «Типография «Новости»
107105. Москва, ул. Фр. Энгельса, 46

профессиональный журнал

ПРОГРАММИСТ

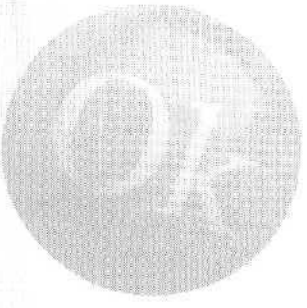
Профессиональный журнал, посвященный исключительно вопросам разработки. Наши **авторы - профессиональные программисты, которые** делятся с читателями «секретами мастерства».

Мы публикуем материалы о самых современных технологиях и **средствах разработки,** статьи а **принципах** и **методах,** теории и практике программирования.

Мы **предлагаем** статьи на самые **разные «программерские» темы,** любого уровня **сложности.**

Подписка

Индекс в каталоге агентства "Роспечать" – 80467, в каталоге "Пресса России" - 45775.



www.programme.ru
info@programme.ru

softline®
www.softline.ru

119991 г. Москва,
ул. Губкина, д. 8
тел.: (095) 232-0023
e-mail: info@softline.ru

Все для разработки ПО

Почему опытные разработчики приобретают нужные для их работы программы в компании SoftLine?

- Их привлекают низкие цены, т.к. компания работает напрямую с вендорами.
- Их привлекает имеющаяся возможность получения демо-версий и обновлений.
- В выборе программ им помогают каталог SoftLine-direct и сайт www.softline.ru.
- Большая часть ассортимента SoftLine для разработчиков недоступна в других компаниях.

Какие этапы разработки охватывает программное обеспечение, поставляемое SoftLine?

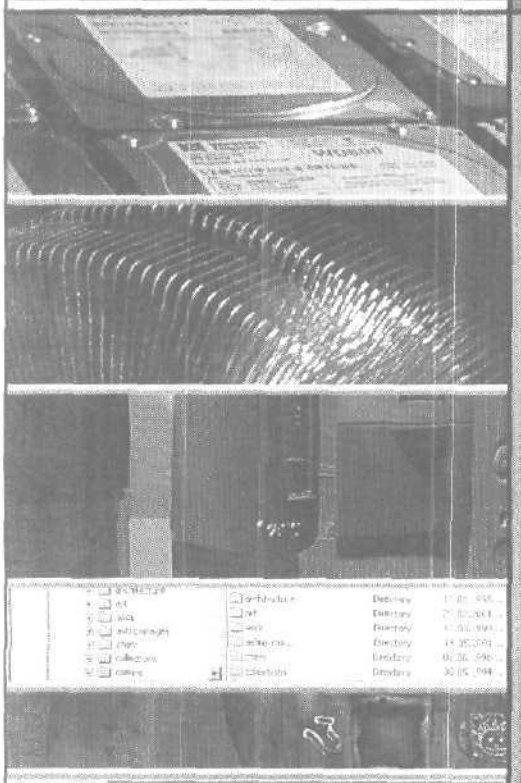
- Проектирование программ (Microsoft, CA/Platinum, Rational, SilverRun, Quest).
- Совместная работа (Centura, Merant, Microsoft).
- Управление проектами (PlanisWare, PlanView, Microsoft).
- Написание кода (среды разработки Allaire, Borland, IBM, Microsoft, компоненты Allround Automation, ComponentOne, Crystal Decisions, Janus, Sitraka, Stingray).
- Оптимизация кода (Compaq, Fuji, Intel, MainSoft, Sun, Sybase, Tenberry),
- Отладка и тестирование (NuMega, Intuitive Systems, Segue).
- Упаковка приложений (InstallShield, Wise Solutions).
- Развертывание и поддержка (Remedy, RoyalBlue, CA, Network Associates).
- Обучение пользователей (Adobe, Allen Communications, click2learn.com, eHelp, Macromedia, Quest, Ulead).

SoftLine — это свобода выбора

Обратившись в SoftLine, вы Б кратчайшие сроки решите проблемы с программным обеспечением. Получив консультацию менеджеров, часть из которых знакома с работой разработчиков не понаслышке (на собственном опыте), вы подберете все необходимое для работы в вашей области - от интегрированной среды RAD - до готовых компонент. При этом мы оставим выбор идеологии разработки за вами - например, для регулярного получения информации о продуктах и технологиях, вы сможете подписаться на Microsoft Developer Network, Sun Developer Essentials или на нашу собственную рассылку компакт-дисков - SoftLine Support Subscription, предоставляющую обновления и демо-версии всех ведущих производителей. Компания SoftLine также поможет вам в выборе обучающих курсов.

Microsoft
Borland
IBM
Sun microsystems
COMPAQ
macromedia
Wise Solutions
Software installations made easy
eHelp
sitraka
<allaire>
NuMega
InstallShield Software Corporation
ComponentOne
SYBASE
InformationAnywhere

ЕЖЕМЕСЯЧНЫЙ
НАУЧНО-ПОПУЛЯРНЫЙ
КОМПЬЮТЕРНЫЙ ЖУРНАЛ





e-mail:
info@hardnsoft.ru

HARD'n'SOFT

МАКСИМАЛЬНО ПОЛНАЯ
И ОБЪЕКТИВНАЯ ИНФОРМАЦИЯ
ДЛЯ ЧИТАТЕЛЕЙ, УВЛЕЧЕННЫХ
КОМПЬЮТЕРНОЙ ТЕХНИКОЙ

В каждом из **номеров** нашего журнала:
- **новости компьютерной индустрии**
- **подробности** о современных и **перспективных технологиях**
- **тесты** и **обзоры аппаратных** я программных **продуктов**
- **интернет** и **мультимедиа,игры** и прикладные ***программы***
- **консультации экспертов,встреч** к t интересными ***людьми***
- CD-приложение с полезными **утилитами**



НАШИ ИНДЕКСЫ:
Hard'n'Soft - 73140,
Hard'n Soft + CD - 26067


BYTE
РОССИЯ
Незаменимый источник технической информации
ря ИТ-профессионалов
• Платформы и решения
• Средства разработки
• Программное обеспечение
• Сети и телекоммуникации
• Технологии связи
• Безопасность
Системным администраторам
Сотрудникам отделов автоматизации
Техническим специалистам
Разработчикам
Оформить подписку или приобрести журнал «BYTE/Россия» можно в офисе службы распространения издательства «СК Пресс». Справки по тел./факсу: (095) 323-1455
Адрес для корреспонденции: 115522 Москва, Пролетарский просп., д. 19, корп 3.
E-mail: deliver@skpress.ru



Подписной индекс по каталогу Агентства «Роспечать» — **81240**
Подписной индекс по каталогу Агентства «Книга-сервис» — **43449**
Интернет-магазин **www.ITbook.ru**

тел.: (095) 142-0571, тел./факс: (095) 145-4519, e-mail: emsdn@rusedit.ru, http://www.rusedit.ru
http://www.microsoft.com/rus/msdn/magazine